VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Návrh a implementace prezentačního software pro mobilní zařízení

Design and Implementation of a Presentation Software for a Mobile Device

2016 Jan Šimeček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Šimeček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Návrh a implementace prezentačního software pro mobilní zařízení
Design and Implementation of a Presentation Software for a Mobile Device

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je návrh a implementace aplikace pro mobilní zařízení, která bude sloužit jako podpůrná prezentační platforma pro příležitosti jako jsou dny otevřených dveří nebo konference. Aplikace bude obsahovat modul pro editaci prezentovaných informací jako jsou výstavní plochy nebo textové informace formou webové aplikace. Další možností je rozšířit aplikaci o jednoduchou hru.

Ve své práci proveďte:
1. Nastudujte možnosti použití frameworku React Native pro vytváření aplikací pro mobilní zařízení.
2. Naimplementujte mobilní aplikaci postavenou na frameworku React Native.
3. Naimplementujte webovou aplikaci pro úpravu dat.
4. Svou aplikaci řádně otestujte a zdokumentujte.
5. Proveďte závěrečné shrnutí.

Seznam doporučené odborné literatury:

[1] Chen, C., Härdle, W., K., Unwin, A.: Handbook of Data Visualization, Springer, ISBN: 978-3540330363
[2] Meier, R.: Professional Android 4 Application Development, Wrox, ISBN: 978-1118102275

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Gaura, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Rád bych tímto poděkoval Ing. Janu Gaurovi, Ph.D. za cenné rady a přípomínky při tvorbě této práce.

Abstrakt

Práce se zabývá návrhem a implementací aplikace pro mobilní zařízení na platformě JavaScript s využitím frameworku React Native. Aplikace bude obsahovat modul pro editaci prezentovaných informací, jako jsou výstavní plochy nebo textové informace, formou webové aplikace s použitím frameworků Django a React. Práce je rozdělena na teoretickou a praktickou část. V teoretické části jsou představeny použité technologie, principy a práce s jednotlivými frameworky. Praktická část je rozdělena na dvě části. V první části je vytvořena webová aplikace pro správu prezentovaných informací, kde jsou na jednoduchých příkladech popsána řešení jednotlivých problémů při implementaci. Část druhá pojednává o tvorbě mobilní aplikace pro systém Android a iOS. Znovu je zde na jednoduchých příkladech vysvětlena implementace konkrétních částí aplikace v rámci obou systémů. Na závěr je provedeno závěrečné shrnutí, zahrnující popis funkcí a použitelnosti výsledných produktů.

Klíčová slova: mobilní aplikace, JavaScript, Python, React Native, React, Django

Abstract

This batchelor thesis is focused on design and implementation of application for mobile devices developed on JavaScript platform in the React Native framework. The application will contain module for management of the presented information like exhibition areas or text information through web application developed using React and Django frameworks. This thesis is divided into theoretical and practical part. The theoretical part presents used technologies, their principles and work with each of them. The practical part is divided into two parts. The first part describes creation of web application for managament of presented information where there are simple examples describing problems that occured during implementation. The second part discusses creation of mobile application for Android and iOS operating systems. There are again simple examples describing problems that occured during implementation of the application for both systems. Finally, there's final summary including description of the functionality and usability of the resulting products.

Key Words: mobile application, JavaScript, Python, React Native, React, Django

Obsah

Seznam použitých zkratek a symbolů

AJAX	–	Asynchronous JavaScript and XML
API	–	Application Programming Interface
DOM	–	Document Object Model
ES5/6	–	ECMAScript 5/6
HTML	–	Hyper Text Markup Language
JSON	–	JavaScript Object Notation
PHP	–	Hypertext Preprocessor
QR Code	–	Quick Response Code
REST	–	Representational State Transfer
SVG	–	Scalable Vector Graphics
UI	–	User Interface (Uživatelské rozhraní)
URL	–	Uniform Resource Locator

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Technologie vývoje mobilních aplikací se neustále vyvíjí, velmi aktuálním tématem jsou právě frameworky umožňující psaní nativních aplikacích pomocí JavaScriptu, mezi které patří React Native. Tyto frameworky poskytují speciální widgety a API[1], umožňující programování zdrojového kódu v JavaScriptu s využitím nativních uživatelských prvků, což zrychluje celý vývoj mobilní aplikace, kdy se programátor nemusí učit novému jazyku a API, ale může využívat již dříve nabytých znalostí. Běh aplikace je tedy srovnatelný s aplikacemi psanými v nativních jazycích a netrpí výkonnostními problémy, jako je tomu například při použití frameworku Apache Cordova, kde aplikace běží ve vloženém Webview. V následujících několika kapitolách bude přiblížen vývoj takovéto aplikace spolu s vývojem podpůrné webové aplikace.

1.1 Cíl práce

Cílem této bakalářské práce je navrhnout a implementovat mobilní aplikaci s využitím JavaScriptového framworku React Native, která bude sloužit jako podpůrná prezentační platforma pro možné události jako jsou dny otevřených dveří nebo konference. Budou zde popsány možnosti frameworku React Native, jeho využití a dostupná API.

Aplikace bude dále obsahovat modul pro editaci prezentovaných informací formou webové aplikace. Budou zde představeny technologie SVG, pro zobrazení vektorové grafiky na webových stránkách a JSON, jakožto způsob zápisu dat, použit pro jejich přenos v rámci aplikace. Dále webový aplikační framework Django psaný v Pythonu, který je použit pro backend webové aplikace a JavaScriptový framework React, jenž obstarává její frontendovu část. Spolu s Reactem bude popsána JavaScriptová knihovna Redux, která uchovává stav celé webové aplikace.

Výsledkem této bakalářské práce bude funkční aplikace pro mobilní operační systémy Android a iOS, zobrazující informace vytvořené prostřednictvím webové aplikace.

[1] API - (Application Programing Interface), jde o sbírku funkcí, tříd a procedur dané knihovny, které může programátor využívat.

2 SVG

SVG (Scalable Vector Graphics) je značkovací jazyk založený na XML, pro kreslení dvourozměrné vektorové grafiky s podporou animací a interaktivity s jednotlivými objekty. Specifikace SVG je otevřený standart vyvíjen organizací World Wide Web Consortium (W3C) od roku 1999. Obrázky SVG spolu s jejich chováním jsou definovány v XML, to znamená, že je nad nimi možné provádět různé operace hledání, indexování, a jinak zpracovávat daný obsah. SVG, podobně jako je tomu u XML, je možné psát ručně v textovém editoru, složitější obrazce jsou však často vytvářeny pomocí specializovaných softwarů. [6]

2.1 SVG a jeho použití na webových stránkách

Použití SVG na webových stránkách bylo dříve limitováno nedostatkem podpory ze strany prohlížečů, konkrétně Internet Explorer do verze 8 nepodporoval zobrazování SVG a verze 8 vyžadovala speciální plugin. V dnešní době je však bezpečné používat SVG ve specifikaci 1.1, která zaručuje plnou podporu všech moderních webových a mobilních internetových prohlížečů.

SVG je možné na stránku vložit několika způsoby. Jedním z nich je přímé zapsání SVG objektů (značek) mezi značky `<svg></svg>` v html dokumentu. Dalším je zobrazení obrázku pomocí značky `<img />` s odkazem na externí soubor s příponou `.svg`.

2.2 Stylování SVG

SVG je možné stylovat pomocí inline atributů nebo použitím kaskádových stylů. SVG objekty používají pro stylování jiné názvy vlastností, kdy například vlastnost `background` se stává vlastností `fill` nebo `border-color` se stává `stroke`. Různé objekty SVG podporují jiné sady daných vlastností, kdy například objekt `line` nepodporuje vlastnost `fill`. Na tento problém je proto nutné dbát při stylování SVG objektů.

```
1 <svg width="400" height="110">
2   <rect width="300" height="100" x="30" y="30" fill="#4bb9f5" stroke="#108cca" stroke-width="1">
3   </rect>
4 </svg>
```

Výpis 2.1: Ukázka vykreslení čtverce s vlastními styly pomocí tagu `<rect />`

3 JSON

JSON (JavaScript Object Notation) je odlehčený formát pro výměnu dat. Jedná se o způsob zápisu dat, snadno čitelný i zapisovatelný člověkem a analyzovatelný strojově. Je založen na podmnožině programovacího jazyka JavaScript. JSON je textový a zcela nezávislý na počítačové platformě, avšak využívá konvence dobře známé programátorům jazyků rodiny C. Tyto vlastnosti jej dělají skvělým kandidátem pro výměnu dat na webu.

3.1 Struktura

Zápis JSONu je zápis JavaScriptu, je tedy sám o sobě už připravenou datovou strukturou pro zpracování. Dá se říct, že JSON je serializovatelnou obdobou datových typů v JavaScriptu. JSON podporuje následující datové typy:

- **JSONString** - textový řetězec
- **JSONNumber** - číslo (celočíselné nebo reálné, včetně zápisu s exponentem)
- **JSONBoolean** - logická hodnota
- **JSONNull** - hodnota null
- **JSONArray** - pole
- **JSONObject** - objekt

Jiné datové typy neexistují, není možné je vkládat přímo. Pokud bychom chtěli uložit například datum, objekt `Date`, je nutné jej nejprve převést na textový řetězec.

3.2 JSON vs. XML

JSON se objevil v době, kdy se pro výměnu dat na webu využíval převážně formát XML. Práce s ním v JavaScriptu byla však mnohdy složitá, bylo nutné používat DOM, řešit přítomnost bílých znaků u prázdných elementů a apod. A ačkoliv JSON nemůže (a ani nemá) konkurovat XML v případě zápisu celých dokumentů a složitých datových struktur, je ideální pro zápis krátkých strukturovaných dat vyměňovaných webovými aplikacemi.

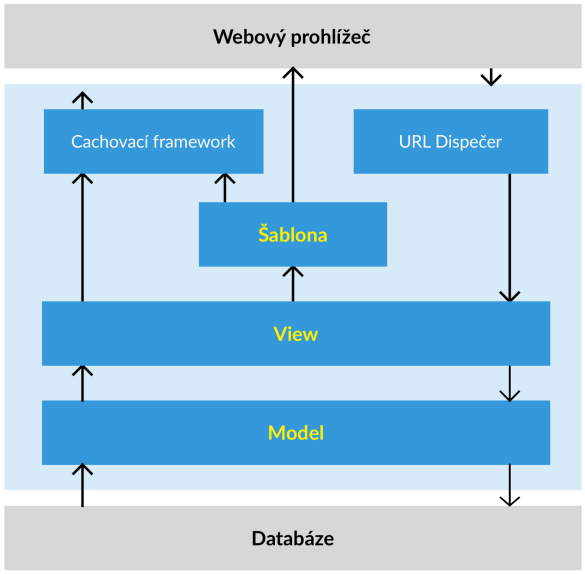
```
1 {
2   "window": {
3     "title": "Titulek",
4     "description": "Main Window",
5     "width": 500,
6     "height": 500
7   }
8 }
```

Výpis 3.1: Ukázka jednoduchého zápisu JSON objektu

4 Django

Django je open-source webový aplikační framework napsaný v Pythonu, který využívá softwarové architektury MVC. Django se zrodilo koncem roku 2003, kdy weboví programátoři Adrian Holovaty a Simon Willison ze zpravodajské společnosti JLawrence Journal-World začali používat Python pro tvorbu aplikací. V červenci 2005 byl framework veřejně vydán pod open-source licencí BSD. V červnu 2008 byla vytvořena nová nezisková organizace Django Software Foundation (DSF), která udržuje vývoj toho frameworku dodnes. [5]

Primární úkol Djanga spočívá v ulehčení procesu vytváření komplexních webových stránek a aplikací využívajících databáze pro ukládání dat. Django klade důraz na znovu použitelnost, rozšiřitelnost a rapidní vývoj. Programovací jazyk Python je používán naskrz celým frameworkem, pro konfigurační soubory, ale i pro datové modely. Django také nabízí dynamicky generované administrativní rozhraní pro správu datových modelů.



Obrázek 4.1: Architektura frameworku Django

4.1 Komponenty

Jádro frameworku Django obsahuje několik důležitých komponent, které výrazně urychlují počáteční vývoj webových stránek. Mezi ty nejdůležitější můžeme zařadit:

- **Objektově relační mapper**, který je prostředkem mezi datovým modelem, reprezentovaným třídami v Pythonu, a relační databází.
- **Systém pro zpracování HTTP požadavků** a zároveň jejich zobrazení.
- **Šablonovací systém**, který využívá koncepty dědičnosti, podobně jako je tomu u objektově orientovaných programovacích jazyků.

- **Samostatný webserver** pro vývoj a testování.
- **Systém pro serializaci**, který produkuje a čte JSON nebo XML reprezentace modelových instancí.
- **Validační systém pro HTML formuláře**, který zároveň překládá hodnoty formuláře na hodnoty vhodné pro uložení do databáze.

4.2 Šablonovací systém

Zpracovaná data je možné zobrazit například ve formátu JSON s využitím funkcí pro serializaci nebo vykreslit do HTML dokumentu prostřednictvím vestavěného šablonovacího sytému. Šablona představuje jednoduchý textový soubor, doplněn o speciální značky, které umožňují například vypsání textu z daného modelu. Modelem rozumíme instanci třídy Context, která se konstruuje ze slovníkového objektu.

Šablonovací systém je velice flexibilní a umožňuje použití vlastních rozšíření. Mezi největší výhody jeho použití patří možnost definice různých bloků uvnitř jednotlivých šablon. Tyto bloky představují jakýsi koncept dědičnosti, kdy můžeme v potomkovi definovat či rozšířit sekce definované v předkovi. Sekce se definují klíčovým slovem `block`

```
1 <div id="container">
2   <div id="sidebar">{% block sidebar %}{% endblock %}</div>
3   <div id="content">{% block content %}{% endblock %}</div>
4 </div>
```

Výpis 4.1: Ukázka použití dědičnosti šablonovacího systému Djanga

4.3 Objektově relační mapování

Jedna z dalších, a nejdůležitějších, součástí jádra frameworku Django je objektově relační mapování. Jednotlivé modely definujeme děděním ze třídy `Model`, která se nachází v modulu `django.db.models`. Jedná se o implementaci návrhového vzoru ActiveRecord, kdy každá instance reprezentuje jeden řádek v databázi a je nad ní možné provádět operace ukládání, mazání a aktualizace voláním příslušných metod přímo nad daným objektem. Každá třída má mimo definici vlastních atributů automaticky vytvořené pole `id`, které slouží k identifikaci konkrétního objektu v databázi. Programátor má na výběr z velkého množství předdefinovaných typů databázových polí, nebo lze také použít i vlastní implementaci.

```
1 from django.db import models
2
3 class Event(models.Model):
4   title = models.CharField(max_length=200, blank=False)
5   content = models.TextField(blank=True)
```

Výpis 4.2: Ukázka definice vlastního modelu v frameworku Django

5 React

React (také nazýván React.js či ReactJS) je open-source[2] JavaScriptová knihovna pro tvorbu uživatelských rozhraní webových stránek pod správou společností Facebook a Instagram. [3]

Byla vytvořena Jordanem Walkem, softwarovým inženýrem společnosti Facebook, který byl ovlivněn technologií XHP, jazykem na bázi PHP pro tvorbu znovupoužitelných komponent psaných v HTML. Její první nasazení nastalo v roce 2011 pro newsfeed na stránkách facebook.com, dále pokračovalo v roce 2012 pro instagram.com a nakonec v květnu 2013 na konferenci JSConf v USA byla představena veřejnosti open-source licencí.

5.1 Hlavní výhody Reactu

V dnešní době moderních webových aplikací máme k dispozici mnoho JavaScriptových knihoven a frameworků. Počínaje jednoduchými knihovnami jako například jQuery nebo Zepto.js, které se převážně zabývají prototypováním základních JavaScriptových objektů a jejich rozšiřováním s cílem zjednodušení práce s DOM[3], či rozsáhlejšími frameworky a knihovnami nabízejícími komplexní řešení. Mezi nejznámější patří AngularJS, React, Ember, Meteor a mnoho dalších.

Zatímco například Angular usiluje o zjednodušení vývoje a testování klientských MVC[4] aplikací, React reprezentuje pouze View část této softwarové architektury, nemá tedy žádné vazby na funkčnost ostatních částí aplikace a je jej velice jednoduché implementovat do nových, či již existujících projektů.

Knihovna React byla vytvořena pro rozsáhlé aplikace s daty, které se v čase mění. Takové aplikace často vyžadují velké množství režie, kdy v případě využití dynamického UI, kde na stránce často přibývá, mizí a mění se obsah jednotlivých elementů, každá tato změna vyžaduje změnu struktury DOMu a jeho následné překreslení. Obecným problémem klasického DOMu je to, že není optimalizovaný pro tyto operace. V případě výskytu velkého množství prvků na stránce (častým příkladem je výpis do tabulky, která může obsahovat až několik tisíc záznamů) by změna jednoho z nich vyžadovala následné překreslení celé stránky, které může trvat i několik sekund. Tato doba je v dnešních moderních aplikacích příliš dlouhá a nepřispívá k pozitivnímu celkovému uživatelskému prožitku. React tento problém řeší aplikací vlastních algoritmů porovnávání změn v DOMu, kdy tyto praktiky označuje jednoduše jako VirtualDOM, který je důležitou částí celé knihovny.

5.2 ECMAScript 2015

ECMAScript 2015, označován zkráceně také jako ES6, je šestá edice skriptovacího jazyka normovaná neziskovou organizací ECMA International, nyní využívána především jeho JavaScriptová

[2] Open-source - volně šířitelný počítačový software s otevřeným kódem

[3] DOM - Document Object Model, objektově orientovaná reprezentace XML nebo HTML dokumentu

[4] MVC - Model View Controller, softwarová architektura, která rozděluje model aplikace, řídící logiku a uživatelské rozhraní do tří nezávislých komponent

implementace. React poskytuje API pro psaní kódu v současné verzi ES5, ale také pro ES6 s použitím tříd. Od verze 0.13 je tento způsob zápisu dokonce doporučován. Využití třídního zápisu komponent v ES6 zpřehledňuje zdrojový kód, který se mnohem více podobá vyšším programovacím jazykům, a tak se stává mnohem srozumitelnějším.

V dnešní době internetové prohlížeče však ještě zdaleka nepodporují všechny prvky syntaxe jazyka ES6. Pro zajištění této podpory je nutné použít Babel transpiler, který převádí syntaxi ES6 na její ekvivalenty psané v ES5.

Poznámka 1 Vzhledem k použití ES6 při implementaci praktické části této bakalářské práce budou i všechny níže zmíněné příklady psané v této verzi jazyka.

5.3 Reaktivní programování

V tradičních JavaScriptových aplikacích jsme nuceni sledovat, jaká data se změnila, a v závislosti na nich provést změny DOMu, aby byl jeho stav aktuální. V Reactu jednoduše vyjadřujeme, jak by měl stav naší aplikace vypadat v jakémkoliv bodě v čase. V případě změny dat React automaticky provede nutné změny uživatelského rozhraní a aktualizace DOMu. Tento proces nám umožňuje programovat aplikaci takovým způsobem, jakoby byla celá stránka znovu překreslena, zatímco React provede aktualizace pouze těch komponent, u kterých byla provedena nějaká změna.

5.4 Virtual DOM

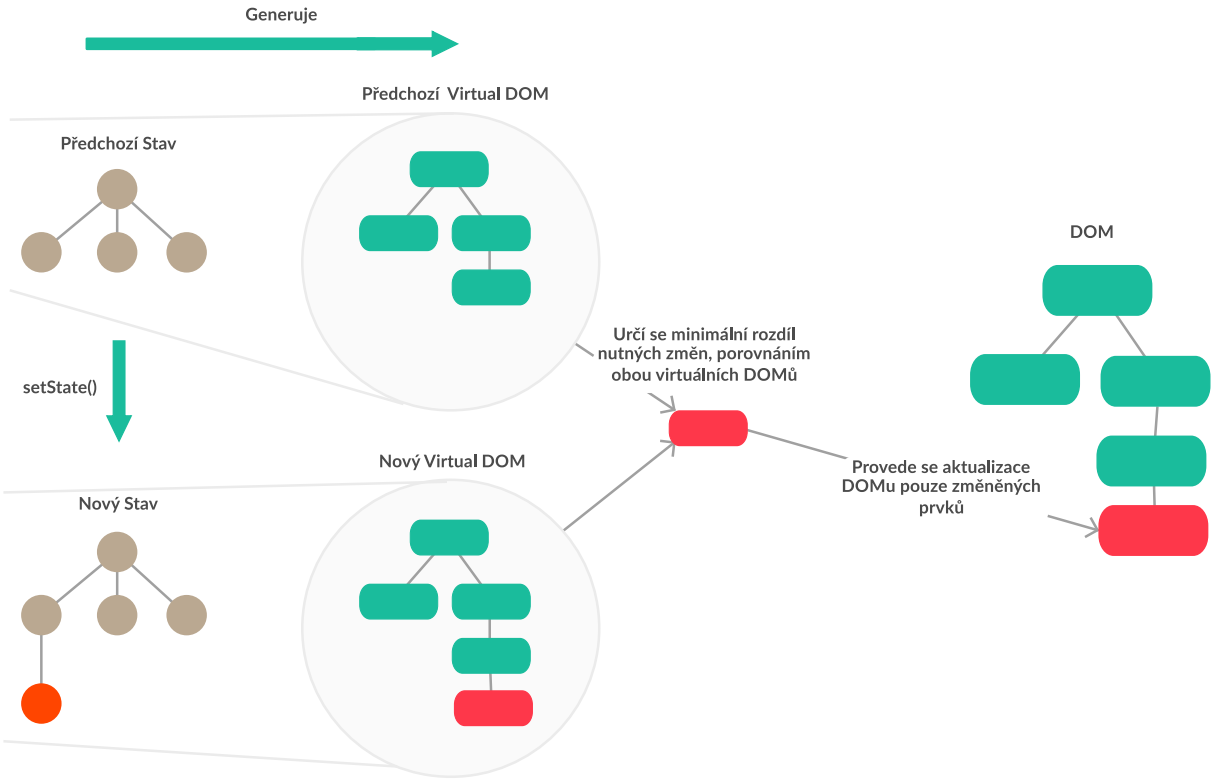
Když je komponenta prvně inicializována, metoda `render` generuje jednoduchou virtuální reprezentaci výsledného pohledu a z této reprezentace produkuje řetězce výsledného kódu, který je následně vložen do DOMu. V případě změny dat, je zavolána metoda `render` znovu. Za účelem provádění co nejefektivnějších aktualizací je vygenerována nová virtuální reprezentace, která se porovnává s předchozí a generuje minimální počet nutných změn, které jsou následně aplikovány na DOM. Tento proces se nazývá Reconciliation (odsouhlasení).

Generování minimálního počtu operací pro transformaci jednoho stromu na jiný je komplexní a důkladně studovaný problém. I ty nejlepší algoritmy mají složitost $O(n^3)$, kdy n je rovno počtu prvků daného stromu. To znamená, že zobrazení 1000 prvků by požadovalo 1 milión porovnání, to je však příliš mnoho a i v dnešní době by na mnoha zařízeních trvalo několik sekund. Z toho důvodu React implementuje neoptimální algoritmy se složitostí $O(n)$ s použitím heuristických metod založených na dvou předpokladech:

1. Dvě komponenty stejné třídy vygenerují dva podobné stromy, zatímco dvě komponenty jiných tříd vygenerují dva různé stromy.

2. Jsme schopni poskytnout unikátní klíče pro prvky, které jsou stabilní napříč různým počtem vykreslení.

Aby mohl React provádět porovnání jednotlivých stromů, musí být nejdříve schopen porovnat dva různé prvky. Existují tři různé příklady, které jsou řešeny.



Obrázek 5.1: Vizualizace výpočtu nového DOMu v rámci metody `render()`

5.4.1 Různé typy prvků

Pokud jsou typy dvou prvků různé, React se k nim chová jako k dvěma různým pod-stromům, první prvek se zahodí a druhý se vloží nebo zpracuje.

Stejná logika je použita v případě vlastních komponent. Pokud jsou komponenty různých typů, React se je ani nesnaží porovnávat a jednoduše první komponentu odstraní a druhou vloží do DOMu. Použití této metody zajišťuje rychlost a přesnost porovnávacího algoritmu. Využívá se principu, kdy je velice nepravděpodobné, že komponenty `<Menu />` a `<Clanek />` mají stejný výstup, nemá tedy cenu se zabývat velkými kusy kódu, ale spíše se soustředit na části, u kterých je podobnost pravděpodobnější. V opačném případě kdy jsou dvě komponenty stejné, je možné předpokládat, že budou mít i podobný výstup a stojí zato je dále porovnávat.

5.4.2 Prvky DOMu

Při porovnávání dvou různých prvků DOMu, React bere v potaz jejich jednotlivé atributy. V případě změny jsou nahrazeny hodnoty pouze těch, které byly změněny. V případě použití inline-stylů (atribut `style`), jsou styly definovány pomocí JavaScriptových objektů klíč-hodnota.

Klíč představuje název vlastnosti ve formátu `camelCase`[5] a hodnota je číslo nebo řetězec. Pak je možné jednoduše nahradit pouze vlastnosti, které byly aktualizovány, což by v případě stylu reprezentovaným řetězcem nebylo možné.

5.4.3 Vlastní komponenty

React považuje dvě vlastní komponenty jako stejné. Vzhledem k tomu, že jednotlivé komponenty jsou stavové, není možné jen tak použít novou komponentu a předchozí zahodit jako to bylo v předchozích případech. Místo toho React zavolá metodu `component[Will/Did] ReceiveProps()` na předchozí, tato komponenta se tak stává funkční, je na ni zavolána metoda `render()` a výstup této metody se porovnává s výstupem nové komponenty.

5.4.4 Seznamy

Porovnání jednotlivých potomků není možné provést v rozumné časové složitosti, React tento problém řeší zavedením nepovinného atributu `key`. Tento atribut představuje unikátní identifikátor daného prvku, je pak možné s využitím hashovací tabulky specifikovat přesuny, vkládání a mazání prvků v časové složitosti $O(n)$. Klíče prvků musí být unikátní pouze vzhledem ke svým potomkům, nikoliv globálně.

5.4.5 Kompromisy

Při vývoji je nutné dbát na proces odsouhlasení, vzhledem k dříve definovaným pravidlům mohou nastat následující stavy, které degradují výkon aplikace:

1. Algoritmus se nesnaží porovnávat pod stromy různých tříd komponent. Pokud máme dvě a více komponent s podobným výstupem, je výhodné tyto komponenty nahradit jednou obecnou.

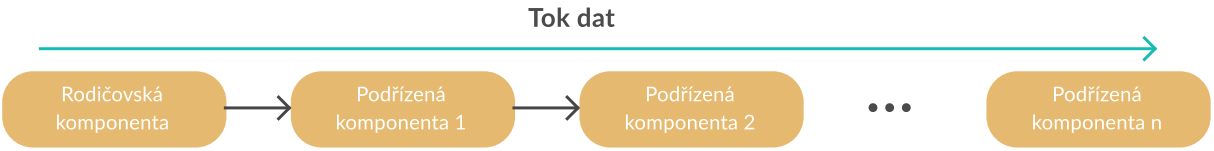
2. Klíče (`key`) by měly být stabilní a unikátní. Nestabilní klíče (například generovány metodou `Math.random()`) mohou vést ke zbytečnému znovuvytvoření potomků a ztrátě jejich stavu.

5.5 Jednosměrný tok dat

React implementuje jednosměrný tok dat, kdy vlastnosti, sada neměnných hodnot, jsou posílány jednotlivým komponentám, které v závislosti na těchto datech provádí, pomocí metody `render`, vykreslování HTML tagů do DOMu. Komponenty nemohou přímo modifikovat jakékoliv vlastnosti, které jim byly poslány. Mezi těmito vlastnostmi se však mohou objevit callback funkce, určené pro modifikaci těchto dat, které daná komponenta může volat. Tento mechanismus je

[5] camelCase - označuje způsob psaní víceslovných frází, kdy jednotlivé mezery mezi slovy jsou vynechány a každé z nich, vyjma prvního, začíná velkým písmenem

také často označován frází *„data proudí dolů, akce nahoru“*, snižuje se přitom režie nutná na správu jednotlivých komponent a celý proces je jednodušší než tradiční data binding. [6]



Obrázek 5.2: Vizualizace toku dat mezi komponentami uvnitř Reactu

5.6 React API a použití

Sada dostupných API, které poskytuje knihovna React, je relativně malá. Kromě nepovinných doplňků pro správu animací, sadu pomocných funkcí pro manipulaci s neměnnými objekty, či knihovnu pro přímou práci s DOMem, zde nalezneme pouze API pro tvorbu jednotlivých komponent. To je také zapříčiněno tím, co jsem již v předchozí kapitole zmínil, že React představuje pouze prezentační část softwarové architektury MVC. I přesto, díky možnostem zanořování jednotlivých komponent, jejich skládání a vzájemné komunikace je toto API více než dostačující.

Použití knihovny je znovu velice jednoduché, React nabízí širokou škálu možností vyhovující kterémukoliv uživateli. Mezi nejjednodušší patří vložení skriptu knihovny do HTML dokumentu, popřípadě i JSX transformeru, který překládá JSX syntaxi do podoby srozumitelné prohlížeči. Mezi další možnosti patří instalace knihovny pomocí správce balíčků npm a její kompilace s využitím dostupných nástrojů, mezi které patří gulp, browserify nebo webpack, o kterých se zmíním v následující kapitole.

5.6.1 JSX jako rozšíření syntaxe jazyka JavaScript

Jedná se o rozšíření syntaxe jazyka JavaScript, určené pro psaní zdrojového kódu jednotlivých komponent s využitím syntaxe podobné XML. Pomocí XML, a zároveň i JSX, lze velice jednoduše definovat stromovou strukturu s různým počtem prvků a jejich atributy, což zjednodušuje psaní výsledného kódu a ten se stává přehlednější než by tomu bylo v případě použití funkcí a objektů.

```
1 class JSXUsage extends React.Component {
2   render() {
3     return (
4       <h1 id="nadpis" className={10 > 2 ? 'success' : 'error'}>Hello JSX</h1>
5     );
6   }
7 }
```

Výpis 5.1: Definice komponenty v React s využitím ES6 a JSX

[6] Data binding - technika, která váže zdroje dat na poskytovatele a spotřebitele a udržuje jejich vzájemnou synchronizaci.

Použití JSX záleží zcela na programátorovi, je však doporučeno. Výsledkem kompilace JSX kódu je JavaScriptový kód. Můžeme tedy volně kombinovat syntaxi JSX s možnostmi základního JavaScriptu. Tento kód je však nutné uzavřít do složených závorek. Pokud se rozhodneme JSX nepoužívat, je nutné html tagy a vlastní komponenty vytvářet pomocí funkce `React.createElement('h1', {id: 'nadpis'})`.

5.6.2 Komponenty a jejich životní cyklus

React je o vytváření modulárních, znovupoužitelných komponent. Tyto komponenty mohou mít vlastní stav, pracovat s vlastnostmi svých rodičů či pouze sloužit pro tvorbu složitější stromové struktury s využitím jiných komponent.

Jako komponentu můžeme považovat část webové stránky (zdrojový kód HTML), která označuje konkrétní funkční prvek uživatelského rozhraní. Jako příklad si můžeme uvést komponentu `TableCell`, která představuje formátovaný výstup dat buňky v tabulce. Tuto komponentu definujeme pouze jednou a pak ji použijeme několikrát s různými hodnotami parametrů (například `data` a `format`). Sada takových komponent může být dále zanořena v další komponentě `TableRow`, a ta zase v další `Table`. Ve výsledné aplikaci nám poté stačí zavolat komponentu `Table` s patřičnými daty a vykreslí se korektně celá tabulka.

```
1  class Table extends React.Component {
2    render() {
3      return (
4        <TableRow>
5          <TableCell data={1.249240} format="%0.2" />
6          <TableCell data={new Date()} format="DD.MM.YYYY" />
7        </TableRow>
8      );
9    }
10 }
```

Výpis 5.2: Ukázka definice vlastních komponent v React

Životní cyklus komponenty lze rozdělit do následujících tří fází:

1. **Připojení** - komponenta je vkládána do DOMu.
2. **Aktualizace** - komponenta je znovu vykreslována s cílem zjistit, zda má být aktualizovaný DOM.
3. **Odpojení** - komponenta je odstraněna z DOMu.

API komponenty poskytuje metody, které umožňují reagovat na změny v kterékoliv fázi jejího životního cyklu. Fáze připojení a odpojení se často využívají pro registraci a odstranění posluchače událostí na HTML dokument a jeho části. Fáze aktualizace slouží pro práci a formátování dat před jejich vykreslením.

5.6.3 Skládání a zanořování komponent

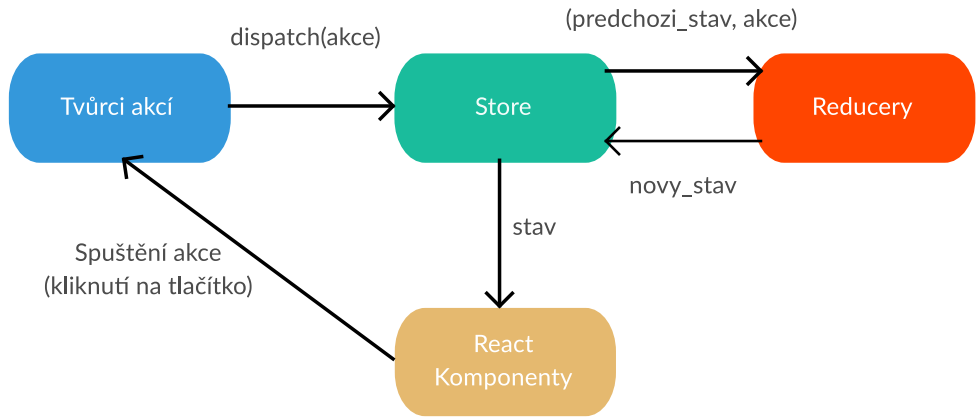
Největší výhodou Reactu je možnost zanořování a skládání jednotlivých komponent. To umožňuje dekompozici řešeného problému na menší části, v tomto případě jednodušší komponenty, které mají menší zodpovědnost za prováděné akce. Tím získáme stejnou výhodu jako například při využití funkcí nebo tříd v jiných programovacích jazycích. Aplikováním těchto aspektů je velice jednoduché si vytvořit vlastní sadu komponent, které jsou znovu využitelné i v jiných aplikacích. Tvoříme tak znovupoužitelný, testovatelný a jednoduše udržovatelný kód.

```
1  class Bootstrap extends React.Component {
2    render() {
3      return (
4        <Row>
5          <Col xs={6}>
6            <Well><h1>Hello Bootstrap</h1></Well>
7          </Col>
8        </Row>
9      );
10   }
11 }
```

Výpis 5.3: Ukázka skládání komponent s využitím prvků knihovny react-bootsrap

6 Redux

V komplexnějších aplikacích, které obsahují značné množství komponent, je docela jednoduché ztratit přehled o celkovém stavu aplikace. Redux tento problém řeší implementací globálního `store`, který uchovává celkový stav naší aplikace na jednom místě. Tento koncept vychází z myšlenek podobného řešení s názvem Flux, vytvořeným společností Facebook, ale vyhýbá se jeho komplexnosti s použitím konceptů funkcionálního programovacího jazyka Elm. [4]



Obrázek 6.1: Tok dat mezi komponentami v Reactu a knihovnou Redux

6.1 Základní principy knihovny Redux

Jedná se o JavaScriptovou knihovnu, která poskytuje řešení pro aplikace, které se chovají konzistentně a mohou běžet v různých prostředích (klient, server nebo nativní) a jsou jednoduše testovatelné. V praxi se jedná o manažera stavu JavaScriptových aplikací. Navíc, díky své implementaci, umožňuje jednoduché řešení pro tvorbu undo/redo, time travel, hot reloading a dalších oblíbených funkcí jinak obtížných na zpracování. Redux lze popsat třemi základními principy, které jsou blíže specifikovány v následujících podkapitolách.

6.1.1 Jeden zdroj dat

Stav naší aplikace je uložen v objektu ve stromové struktuře, který je součástí jediného store. Toto řešení umožňuje jednoduchou tvorbu univerzálních aplikací, jelikož celý stav je možné serializovat a uložit na server bez jakékoliv práce navíc. Jediný store také zjednodušuje ladění výsledné aplikace a umožňuje persistenci jednotlivých stavů aplikace, tento proces dělá funkce, tradičně složité na implementaci jako undo/redo, triviální.

6.1.2 Stav je pouze pro čtení

Jediným řešením, jak zajistit změnu stavu aplikace je vyslání akce (action). To zajišťuje, že žádné view, ani callback funkce síťových volání nemohou přímo měnit stav aplikace. Místo toho

jsou nuceny uvést, jaká změna se má provést a zavolat příslušnou akci. Protože všechny změny stavu jsou centralizované a dějí se jedna po druhé ve striktním pořadí, nevyskytují se zde žádné nepatřičné podmínky, na které musíme dávat pozor.

6.1.3 Změny jsou prováděny čistými funkcemi

Jakým způsobem se stav mění, specifikujeme pomocí čistých (pure) funkcí zvaných reducer. Reducer je čistá funkce, která jako první argument bere předchozí stav aplikace, jako druhý vykonávanou akci a vrací nový stav. Nikdy neměníme obsah předchozího stavu přímo, funkce vždy vrací nový objekt. S rostoucím rozsahem vyvíjené aplikace můžeme reducer rozdělit na několik menších spravujících určité části aplikace. Vzhledem k tomu, že reducer je pouze funkce, je možné kontrolovat pořadí volání těchto funkcí, předávat dodatečná data, či dokonce vytvářet obecné reducery pro řešení častých úkonů jako například stránkování.

6.2 Akce

Akce jsou nositelé informací, které posílají data z aplikace do store. Jedná se o jediný zdroj informací pro store. Akce voláme pomocí `store.dispatch()`. Jedná se o čisté funkce, které vrací JavaScriptové objekty s povinnou vlastností `type`, ta indikuje, jaká akce je právě prováděna. Typ akce by měl být ideálně definován jako řetězcová konstanta. Další vlastnosti objektu jsou čistě na programátorovi, často se jedná o samotné informace, které budou vloženy do daného store. Pokud například máme jednoduchý todo list a chceme vytvořit akci, která vloží novou položku do todo listu, tak tato akce by mohla vypadat nějak takto:

```
1 const ADD_TODO = 'ADD_TODO';
2
3 function addTodo(text) {
4   return {
5     type: ADD_TODO,
6     text: text
7   }
8 }
```

Výpis 6.1: Ukázka akce pro manipulaci store v Redux

Je doporučeno předávat akcemi co nejmenší počet dat, v případě získání konkrétního prvku z pole je určitě lepší poslat `index` nebo `id` dané položky v poli, než celý objekt.

6.3 Reducery

Akce nám říkají, že se něco stalo, nepopisují však jak se změní stav aplikace v závislosti na poslaných datech. To je práce pro reducer. Jak již bylo řečeno, reducer je čistá funkce, která přijímá dva argumenty: předchozí stav aplikace a akci. Při prvním zavolání reducer vrací hodnotu stavu `undefined`, je proto důležité v tomto případě poskytnout výchozí stav aplikace. Reducer

provádí změny stavu aplikace v závislosti na získaných akcích, v případě neznámé akce musí vždy vrátit předchozí (nezměněný) stav aplikace. Reducer vždy vrací nový objekt, při tvorbě nových reducerů, je proto nutné dbát na to, abychom pouze neměnily hodnoty předchozího stavu, ale vždy vraceli stav nový. Tento princip umožňuje JavaScriptu znovu použít stará nezměněná data při tvorbě nového stavu aplikace a celý proces je tak velice rychlý, což napomáhá celkovému výkonu aplikace. Pokud bychom pokračovali v předchozím příkladu implementace akcí pro todo list, náš reducer pro obsluhu této akce, s využitím syntaxe standartu ECMAScript 2015, by mohl vypadat nějak takto:

```
const initialState = { todos: [] };

const todos = (state = initialState, action) => {
  switch (action.type) {
    case ADD_TODO:
      return {
        todos: [
          ...state.todos,
          { text: action.text }
        ]
      };
    default:
      return state;
    }
};
```

Výpis 6.2: Ukázka reduceru pro obsluhu akce vkládání položky do todo listu

6.4 Store

V předchozích dvou kapitolách jsem zmínil akce, které říkají co se stane, a Reducery, které řeší, jak se data změní v závislosti na dané akci. Store je objekt, který oba tyto procesy spojuje. Obsahuje celý stav aplikace, umožňuje jeho změnu voláním funkce `dispatch(akce)` a poskytuje přístup k jednotlivým reducerům. Vytváří se pomocí funkce `createStore()`, ta bere jako argument reducer. Pomocí funkce `combineReducers()` je možné kombinovat více reducerů do jednoho store. Je důležité si pamatovat, že JavaScriptové aplikace mohou mít neomezené množství reducerů, ale vždy jen jeden store.

7 React Native

React Native je JavaScriptová knihovna, poskytující API pro tvorbu aplikací na nativních mobilních platformách s využitím JavaScriptu a frameworku React. Tato knihovna byla představena 26. března 2015. Je vyvíjena a udržována, podobně jako je tomu u knihovny React, společností Facebook. Cílem React Native je poskytnout efektivní vývoj mobilních aplikací pro platformy iOS a Android. [2]

Celá knihovna je založena na úspěchu Reactu, kdy i dostupné API je mu velice podobné. Toto API je navíc doplněno o platformě specifické funkce, kterými jsou například práce se souborovým systémem, přístup ke kameře zařízení apod.

7.1 Instalace frameworku a podpora

React Native lze nainstalovat pomocí správce balíčků `npm`, celý proces instalace a seznam požadavků je důkladně popsán na oficiálních stránkách. Podpora operačního systému OS X je nejvyšší prioritou při vývoji této knihovny. Jeho použití je ostatně také nutné pokud chceme vyvíjet aplikace pro iOS. Existuje, však experimentální podpora pro vývoj Android aplikací na platformách Windows a Linux.

7.2 Nativní komponenty

React Native poskytuje přístup k nativním komponentám dané platformy. V případě iOS mezi ně mimo jiné patří `NavigatorIOS` a u Androidu se jedná například o `Drawer`. Tyto nativní komponenty vzhledově bez problémů zapadají do zbytku ekosystému dané platformy a zároveň udržují vysokou kvalitu výsledného produktu.

V aktuální verzi 0.21, ke dni 21. března 2016, React Native nabízí velké množství dostupných komponent pro jednotlivé platformy. Tyto komponenty můžeme rozdělit do čtyř kategorií:

1. **iOS komponenty** - specifické pro operační systém iOS. Názvy těchto komponent často obsahují příponu „IOS“. (`NavigatorIOS`, `PickerIOS`, ...).

2. **Android komponenty** - specifické pro operační systém Android. Znovu, podobně jako je tomu u komponent pro iOS, název těchto komponent také obsahuje příponu „Android“. (`ProgressBarAndroid`, `PullToRefreshViewAndroid`, ...).

3. **Obecné nativní komponenty** - dostupné pro oba systémy. Jedná se spíše o obecnější komponenty. (`WebView`, `Text`, ...).

4. **Komponenty psané v JavaScriptu** - velmi často poskytují řešení pro obě platformy. Jedná se o komponenty psané v JavaScriptu. Výhodou je použití jednotného API při tvorbě uživatelských rozhraní obou platforem, mezi nevýhody může patřit degradace výkonu aplikace, zvlášť v případě, že daná komponenta využívá animace. (`Navigator`).

React Native také poskytuje API pro vytváření vlastních nativních komponent. Tento proces je určen zkušenějším vývojářům. Jeho princip spočívá v „obalení“ nativního modulu speciálním API, které umožní komunikaci s jeho protějškem psaném v JavaScriptu.

V případě, že komponenty, které React Native poskytuje, nám nestačí a nechceme si vytvářet vlastní, máme na výběr z nepřeberného množství komponent vytvořených komunitou programátorů používajících React Native. Tyto komponenty jsou k dispozici prostřednictvím správce balíčků `npm`.

```
class HelloNavigator extends React.Component {
  render() {
    return (
      <NavigatorIOS
        style={styles.container}
        initialRoute={{
          component: EventsScreen,
          title: 'Events',
          passProps: { myProp: 'foo' }
        }} />
    );
  }
}
```

Výpis 7.1: Ukázka použití komponenty NavigatorIOS v React Native

7.3 Platformově specifický kód

Při vývoji multiplatformních aplikací můžou nastat situace, kdy je nutné psát odlišný kód pro každou platformu či provádět určité operace pouze na jedné z nich. React Native nabízí API pro řešení těchto problémů.

7.3.1 Platformově specifické názvy souborů

React Native rozlišuje soubory v případě, že se v jejich názvu nachází rozšíření `.ios.` nebo `.android.` a načte správný soubor odpovídající dané platformě. Tento způsob například umožňuje psaní dvou stejných komponent s různými implementacemi pro jednotlivé systémy a můžeme tak například využít platformově specifických komponent zmíněných v kapitole 7.2.

7.3.2 Modul Platform

React Native mimo podpory platformově specifických názvů souboru také nabízí modul pro detekci platformy přímo v zdrojovém kódu. Tyto funkce lze použít v případech, kdy ovlivňujeme pouze malou část zdrojového kódu a vytvoření nové komponenty by tak bylo zbytečné. U systému android může být tento modul také použit pro detekci verze operačního systému, ve kterém aplikace běží.

```
const styles = StyleSheet.create({
  container: {
    marginTop: (Platform.OS === 'ios') ? 64 : 56,
  }
});
```

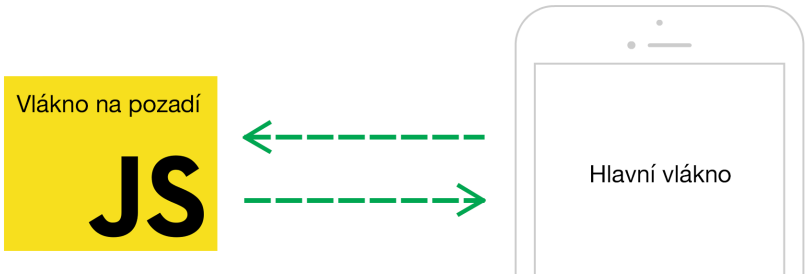
Výpis 7.2: Ukázka použití modulu Platform v React Native

7.4 Asynchronní zpracování

Všechny operace mezi JavaScriptem a nativní platformou jsou dávkovány a prováděny asynchronně, navíc mohou nativní moduly utilizovat vlastní sadu vláken. To znamená, že React Native může dekódovat obrázky mimo hlavní vlákno, ukládat data na disk na pozadí nebo počítat rozměry rozložení uživatelského rozhraní bez nutnosti blokování hlavního vlákna 7.1. Díky tomuto řešení jsou React Native aplikace přirozeně plynulé a responzivní. Celá komunikace je zároveň plně serializovatelná, což umožňuje podporu nástrojů pro ladění JavaScriptového kódu prostřednictvím Google Chrome Developer Tools. Při spuštění React Native aplikace, může být kód zpracován ve dvou různých prostředích:

- V iOS simulátorech a zařízeních, Android emulátorech a zařízeních React Native používá **JavaScriptCore**, jedná se o JavaScriptový engine, který je používán v Safari.
- Při použití nástrojů pro ladění, je veškerý JavaScriptový kód vykonáván uvnitř samotného prohlížeče Chrome na JavaScriptovém enginu **V8** a komunikuje s nativním kódem prostřednictvím web soketů.

Přestože jsou obě prostředí velice podobná, může se stát, že při vývoji narazíme na nějaké nekonzistence. Neměly bychom se proto spoléhat na funkce specifické pro daný JavaScriptový engine.



Obrázek 7.1: Komunikace JavaScriptu s nativním modulem v React Native

7.5 Dotekový systém

Rozpoznávání doteků a gest na mobilních zařízeních je mnohem komplikovanější proces, než je tomu na webu. Dotek může projít několika fázemi, než dokáže systém určit, zda se jedná o gesto, posun či jednoduchý dotek. React Native obsahuje výkonný systém pro rozpoznávání těchto doteků, poskytuje dvě komponenty `TouchableHighlight` a `TouchableOpacity`, kterými můžeme „obalit“ jakékoliv jiné komponenty. Tyto komponenty poskytují metodu `onPress()`, jejíž chování je správné i v případě použití komponenty `ScrollView`, nebo uživatelských rozhraní s velkým množstvím komponent.

React Native také poskytuje systém, pro rozpoznávání gest a více souběžných doteků zároveň zvaný `PanResponder`. Tento systém umožňuje programátorům přístup k jakési abstrakci nativního systému pro rozpoznávání gest. S využitím dostupných metod, které toto API poskytuje, je možné reagovat takřka na jakékoliv typy doteků a gest, což umožňuje tvorbu různých uživatelských rozhraní nebo jednoduchých her.

7.6 Flexbox a stylování

Stylování aplikací v React Native je mnohdy jednoduší než v nativních jazycích. To je způsobeno použitím `flexbox` modelu pro rozvržení rozhraní a ostatními vlastnostmi pro stylování jako jsou například `padding` a `margin` propůjčené z kaskádových stylů webových stránek.

Styly definujeme vytvářením JavaScriptových objektů pomocí funkce `StyleSheet.create()`. Tato funkce poskytuje optimalizovaný mechanismus pro tvorbu stylů. Ujišťuje se, že všechny hodnoty vlastností jsou neměnné, a to jejich transformací na prostá čísla, které jsou referencemi pro skutečné styly ve vnitřní tabulce systému. Jejich definicí na konci souborů také zajistíme, že se styly vytvoří pouze jednou, nikoliv při každém zavolání metody `render`. V dalších voláních se na objekt stylů odkazuje jeho referencí. Styly můžeme přiřadit jednotlivým komponentám pomocí atributu `style`.

```
1 <View style={styles.container} />
2
3 const styles = StyleSheet.create({
4   container: {
5     flex: 1,
6     padding: 15,
7     backgroundColor: '#bbb'
8   }
9 });
```

Výpis 7.3: Ukázka definice stylů v React Native

7.7 Polyfills

React Native nabízí mimo jiné rozšíření JavaScriptového API v podobě implementace univerzálních webových standardů nazývané Polyfills. Tyto standarty se chovají stejně na všech platformách. Mezi ty nejdůležitější můžeme zařadit:

- **Fetch** - API pro správu HTTP požadavků. V současné době stále vyvíjeno pro použití ve webových prohlížečích a momentálně dostupné pouze v prohlížeči Google Chrome.
- **WebSocket** - protokol, který nabízí plně duplexní komunikační kanál nad protokolem TCP.
- **Podpora barevných formátů** - React Native nabízí podporu pro širokou škálu barevných formátů, od modelu HSL po zápis pomocí hexadecimálního čísla.
- **Geolokace** - API pro určení geografické polohy s využitím GPS a ostatních senzorů samotného zařízení.
- **Timers** - mimo výchozí sadu funkcí pro tvorbu časovačů a intervalů, implicitně dostupných v JavaScriptu, React Native poskytuje speciální objekt `InteractionManager`, který umožňuje provedení akcí, až jakmile skončí veškeré dosavadní interakce se zařízením. Tento způsob zajišťuje omezení ztráty plynulosti aplikace v případě provádění složitějších výpočtů.

8 Návrh webové aplikace

Cílem této bakalářské práce je vyvinout jednoduchou webovou aplikaci s daty uloženými na serveru, která řeší správu událostí a map, které následně slouží jako datový zdroj pro mobilní aplikaci. Možnost editace těchto dat je dostupná pouze administrátorovi a odpadá tedy nutnost řešení správy uživatelů.

8.1 Specifikace zadání

Po podrobnější analýze a konzultaci s vedoucím práce bylo zadání specifikováno následujícími několika body:

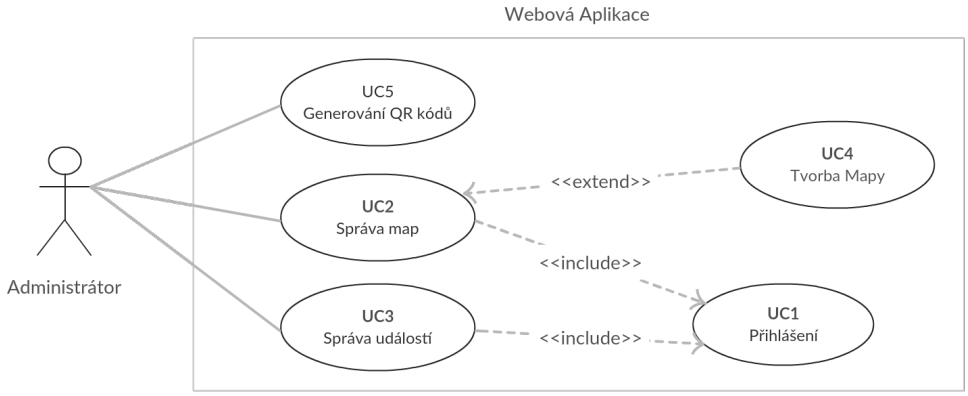
1. Webová Aplikace poskytne nástroje pro vytváření a správu map. Zároveň umožní administrátorovi přiřadit akce jednotlivým objektům na mapě, které při kliknutím či dotyku zobrazí další textové informace.
2. Webová Aplikace dále umožní administrátorovi spravovat události, jejich obsah a možnost propojení s dříve vytvořeními mapami. Mimo jiné umožní tvorbu jednoduchých soutěží v rámci dané události.
3. Webová Aplikace bude poskytovat veřejné API pro přístup k datům z mobilní aplikace.
4. Webová Aplikace bude poskytovat jednoduché nástroje pro generování QR kódů čitelných mobilní aplikací.
5. Webová Aplikace by měla být navržena s ohledem na možná rozšíření.
6. Serverová část (backend) bude implementována s využitím webového aplikačního frameworku Django v jazyce Python.
7. Klientská část (frontend) bude implementována s využitím JavaScriptové knihovny React.

8.2 Návrh uživatelského rozhraní

Vzhledem k jednoduchosti celé webové aplikace bude mít výsledná webová stránka pět hlavních stránek pro: přihlášení, správu událostí, správu map, generátor QR kódů a editor mapy. Tyto stránky budou na vhodných místech doplněny o modální okna obsahující formuláře pro editaci a vkládání nových dat, které budou zpracovávány asynchronně. Tento způsob návrhu uživatelského rozhraní eliminuje mnohdy časté a nepříjemné načítání stránek spojené se stránkami dedikovanými formulářům pro správu dat.

8.2.1 Přihlášení

Stránka pro přihlášení bude obsahovat pouze jednoduchý formulář se vstupními poli pro uživatelské jméno, heslo a zaškrtávací políčko pro zapamatování přihlášení.



Obrázek 8.1: Diagram případů užití v rámci webové aplikace

8.2.2 Správa událostí

Hlavním prvkem této stránky je tabulka zobrazující jednotlivé události. Každý řádek tabulky obsahuje, mimo titulek a data začátku a konce události, také tlačítka pro editaci nebo odstranění daného záznamu. Vpravo nahoře dále najdeme tlačítko, které zobrazí modální okno s formulářem pro vytvoření nové události.

8.2.3 Správa map

Stránka pro správu map je velice podobná stránce pro správu událostí. Zase zde dominuje tabulka zobrazující všechny již vytvořené mapy. Jediným rozdílem oproti předchozí stránce je přidání nového akčního tlačítka jednotlivým záznamům pro vytvoření či úpravu mapy. Po kliknutí na toto tlačítko jsme přesměrováni na novou stránku s editorem.

8.2.4 Generátor QR kódů

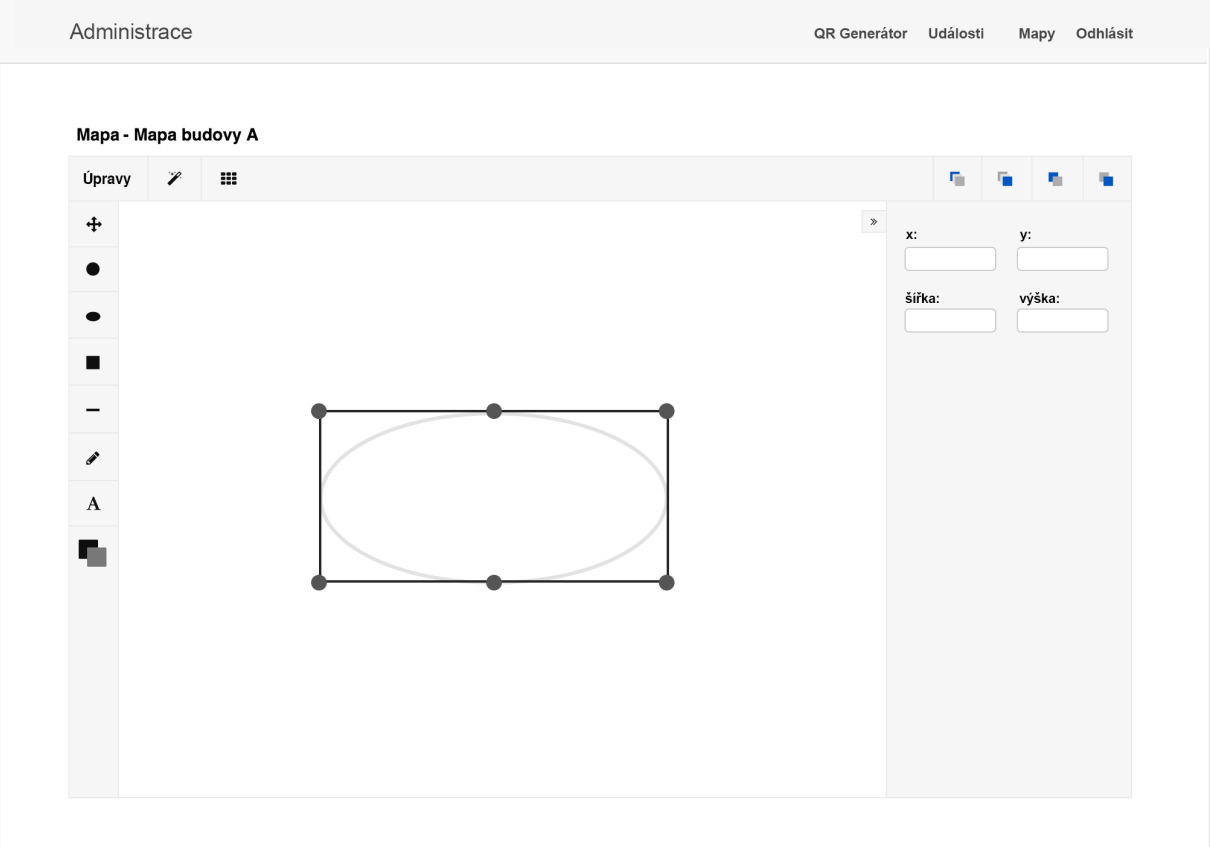
Jedná se o nejjednodušší stránku v rámci celé aplikace. Tato stránka bude obsahovat pouze vstupní pole, pro jehož obsah aplikace vygeneruje daný QR kód, který se zobrazí nad tímto polem. Dalším prvkem bude tlačítko, které po kliknutí stáhne vygenerovaný QR kód v podobě obrázku.

8.2.5 Editor mapy

Editor mapy patří mezi nejsložitější prvky celé webové aplikace. A to jak ze strany implementace, tak z pohledu návrhu uživatelského rozhraní. Inspirace pro rozvržení jednotlivých prvků byla čerpána z grafických programů pro tvorbu 2D vektorové grafiky. [1]

Editor se skládá z plátna, horního ovládacího panelu, který na levé straně obsahuje rozevírací menu s dostupnými akcemi, tlačítko pro definování výchozích stylů a tlačítko pro úpravu velikosti sítě plátna. Na pravé straně se nachází čtyři tlačítka, která slouží pro řazení jednotlivých objektů mezi sebou. Dále se na levé straně vyskytuje levý ovládací panel, který obsahuje veškeré nástroje

pro kreslení na plátno. Nakonec, v případě, že máme vybraný nějaký objekt, napravo najdeme vysouvací menu zobrazující parametry daného objektu a jejich hodnoty, obrázek 8.2.



Obrázek 8.2: Návrh uživatelského rozhraní editor mapy

8.3 Design uživatelského rozhraní

Design uživatelského rozhraní vychází z jeho návrhu v předchozích kapitolách. Proces výběru designu webových stránek závisí především na jejich využití, kdy například reklamní stránka se snahou zaujmout určitý druh zákazníků bude vypadat jinak než webová aplikace, která dbá především na její informativní hodnotu a měla by tedy klást co největší důraz na přehlednost.

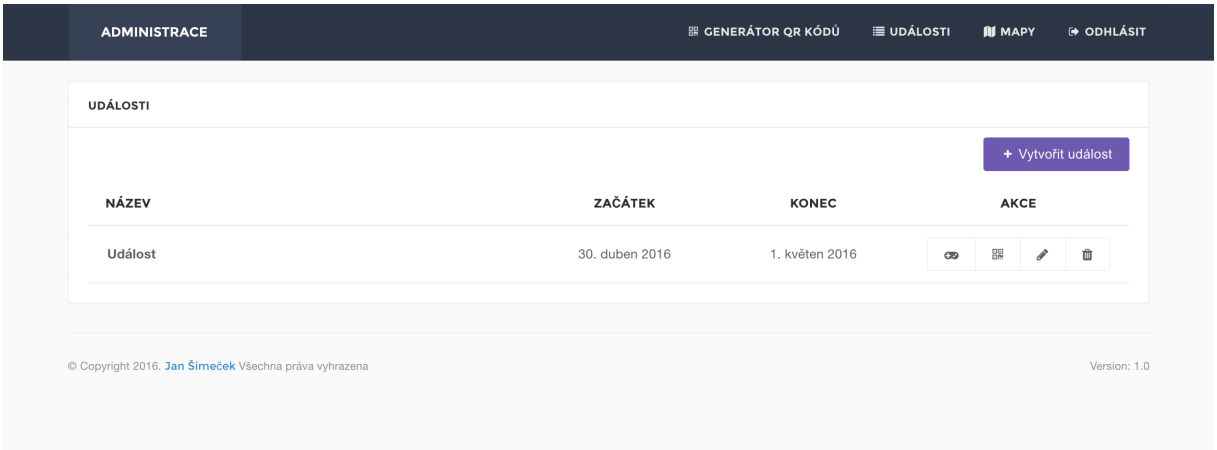
Jako základ pro design a realizaci webové aplikace byla použita knihovna bootstrap, která poskytuje syntaxi a styly pro základní funkční prvky webových stránek jako jsou navigace, modální okna či grid systém. Využití této knihovny výrazně urychlilo vývoj aplikace, která byla posléze doplněna o vlastní implementace nových funkcí a celkové přepracování vzhledu již existujících prvků.

8.3.1 Přihlášení

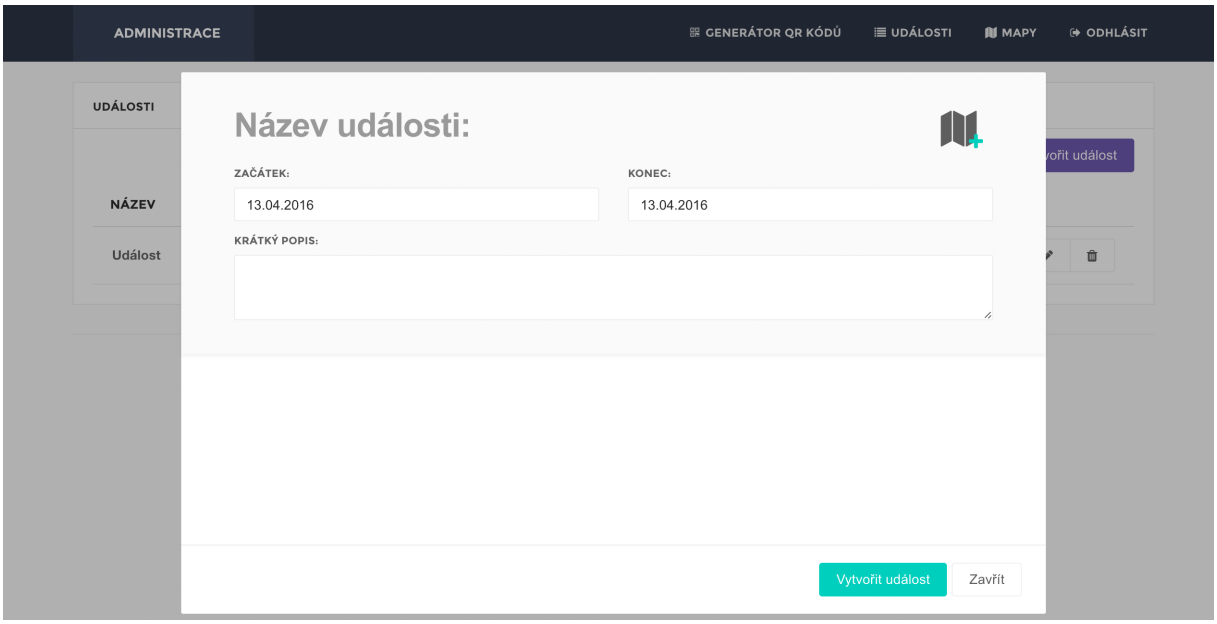
V případě této stránky je design shodný s jeho návrhem, nebylo tedy nutné při realizaci přidávat nové prvky pro zajištění funkcionality přihlášení.

8.3.2 Správa událostí

Obrázky 8.3 a 8.4 představují design stránky pro správu událostí a ukázku jednoho z mnoha modálních oken. V tomto případě se jedná o okno obsahující formulář pro vytváření nových záznamů. Stránka pro správu událostí byla oproti jejímu návrhu doplněna o patičku, zobrazující informace o copyrightu a aktuální verzi webové aplikace.



Obrázek 8.3: Design stránky pro správu událostí



Obrázek 8.4: Design modálního okna pro vytváření událostí

8.3.3 Správa map

Stránka pro správu map se velice neliší od jejího návrhu. Stále zde dominuje tabulka zobrazující jednotlivé mapy. Mezi provedené změny patří texty akčních tlačítek, které byly nahrazeny ikonami s cílem redukce jejich velikosti a také, podobně jako je tomu u stránky pro správu událostí, zde najdeme patičku zobrazující další informace.

8.3.4 Editor mapy

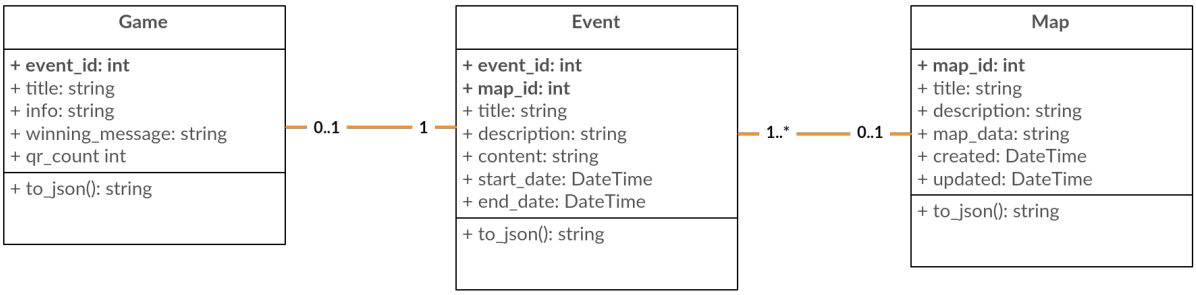
Design stránky editoru mapy znovu odpovídá jejímu návrhu, vyjma přidání patičky, která podobně jako tomu bylo u předchozích stránek, zobrazuje další užitečné informace o webové aplikaci.

8.4 Doménový model

Analýza specifikace zadání v kapitole 8.1 odhalila tři entity: mapa a událost a hra (soutěž). Mezi tyto entity bychom mohli dále zařadit další entitu administrátor či uživatel, pro správu administrátorských účtů. Nicméně vzhledem k tomu, že tato aplikace v serverové části využívá systém Django, který implicitně poskytuje nástroje pro správu uživatelů, budou pro správu administrátorských účtů využity tyto nástroje namísto vlastní implementace.

Dále vzhledem k využití JavaScriptové knihovny React pro klientskou část aplikace, každá entita navíc obsahuje metodu `to_json()`, které serializuje data dané entity do formátu JSON pro jejich přenos mezi serverem a klientem. Názorná představa výsledného doménového modelu je vyobrazen diagramem na obrázku 8.5.

Poznámka 2 Zatímco jsou jednotlivé entity pojmenovány českými názvy, v doménovém diagramu jsou reprezentovány jejich anglickými ekvivalenty. Toto řešení je z důvodu zachování stejného jazyka (angličtiny) napříč implementací, kde i názvy proměnných a objektů jsou psány anglickými názvy.



Obrázek 8.5: Doménový model webové aplikace

9 Implementace webové aplikace

Jak již bylo dříve řečeno, aplikace se skládá ze dvou víceméně oddělených částí, které spolu komunikují prostřednictvím předem definovaných rozhraní. Tato architektura má tu výhodu, že umožňuje tvorbu dynamického uživatelského rozhraní bez nutnosti zásahu do serverové části.

Serverová část je napsána v jazyce Python s využitím webového aplikačního frameworku Django. Klientská část využívá knihovny React pro tvorbu uživatelského rozhraní a knihovnu Redux pro správu stavu aplikace v rámci knihovny React. Komunikace mezi oběma částmi probíhá prostřednictvím technologie AJAX, kde je výměna řešena s využitím formátu JSON.

9.1 Serverová část

Ve 4. kapitole jsem se zmínil o základních konceptech frameworku Django. V následujících kapitolách proto nebudu suplovat dokumentaci toho frameworku, ale pokusím na konkrétních příkladech vysvětlit způsob řešení jednotlivých problémů v rámci implementace této webové aplikace.

9.1.1 URL

Django poskytuje API pro vytváření URL schémat, které nám umožní vytvářet prakticky jakoukoliv strukturu URL definujících webovou stránku. Každé URL má přiřazený jeden view, který reprezentuje Controller v softwarové architektuře MVC. Jednotlivé URL definujeme pomocí regulérních výrazů, kde s využitím speciálních značek můžeme definovat proměnné, s kterými lze dále pracovat v přiřazeném view.

V případě naší aplikace využíváme pro komunikaci se serverem princip architektury REST, která jednoduše umožňuje číst, editovat a mazat záznamy, identifikované pomocí pevně dané cesty (URL) a jednoduchých HTTP volání s využitím HTTP metod. Struktura URL naší webové aplikace tedy zároveň dodržuje doporučené konvence v případě tvorby REST aplikací. Mezi definované URL patří:

- `/events` - Zobrazí stránku pro správu událostí a zároveň při HTTP volání metodou `POST` vytváří novou událost s daty obsaženými v těle tohoto dotazu.
- `/events/<ID>` - Pomocí HTTP metod `GET`, `PUT` a `DELETE` jsme schopni získat, editovat či smazat událost s daným ID.
- `/maps` - Podobně jako je tomu u událostí, i toto URL zobrazí stránku pro správu map a zároveň při HTTP volání metodou `POST` vytváří novou mapu s daty obsaženými v těle tohoto požadavku.
- `/maps/<ID>` - Pomocí HTTP metod `GET`, `PUT` a `DELETE` můžeme získat, editovat či smazat mapu s daným ID.

- **/maps/<ID>/editor** - Zobrazí editor mapy pro mapu s daným ID.
- **/logout a /login** - Tyto URL slouží k zpracování akce přihlášení a odhlášení uživatele.
- **/api/v1/** - Poskytuje veřejné API pro přístup k datům webové aplikace. Více o tomto API se dozvíme v kapitole o implementace mobilní aplikace.

```
1 urlpatterns = [
2   # /events/
3   url(r'^$', views.index, name='events'),
4   # /events/[event_id]
5   url(r'^(?P<event_id>[0-9]+)', views.rest, name='events_rest')
6 ]
```

Výpis 9.1: Definice URL pro správu událostí

9.1.2 Model

Každý model webové aplikace je potomkem třídy `Model`, která se nachází v modulu `django.db.models`. Jelikož kvůli komunikaci s klientskou částí potřebujeme data serializovat do JSONu, každý model zároveň obsahuje implementaci funkci `to_json()`, která vrací slovníkový objekt.

```
1  class Event(models.Model):
2    title = models.CharField(max_length=200, blank=False, null=False)
3    description = models.CharField(max_length=500, blank=False, null=False)
4    content = models.TextField(blank=True, null=True)
5    start_date = models.DateTimeField('Start date', blank=False, null=False)
6    end_date = models.DateTimeField('End date', blank=False, null=False)
7    map = models.ForeignKey(Map, blank=True, null=True)
8
9    def has_game(self):
10     if(hasattr(self, 'game')):
11       return self.game
12     return None
13
14   def __str__(self):
15     return self.title
16
17   def to_json(self):
18     return {
19       'id': self.id,
20       'title': self.title,
21       'description': self.description,
22       'content': self.content,
23       'startDate': self.start_date,
24       'endDate': self.end_date,
```

```
25        'mapId': self.map.id if self.map else None,
26        'game': self.game.to_json() if self.has_game() else None
27    }
```

Výpis 9.2: Implementace datového modelu pro entitu událost

9.1.3 View

Jak již bylo zmíněno v předchozí kapitole 9.1.1 každé URL má přiřazeno jeden view, který reprezentuje Controller softwarové architektury MVC. View definujme jako funkci, která přijímá parametr `request`, jenž je objektovou reprezentací přijímaného HTTP požadavku. V případě naší aplikace, konkrétněji části pro správu událostí, definujeme následující dva view:

- **`index`** - Vrací html dokument se seznamem událostí a map ve formátu JSON pro jejich další zpracování s využitím knihovny React.
- **`rest`** - Definován nad URL s parametrem ID. V závislosti na přijaté HTTP metodě provádí editaci, mazání nebo vrací záznam s daným ID.

Abychom byli schopni určit, o jakou akci se jedná, v rámci daného view, musíme nejdříve zjistit, jaká metoda byla přijatá v HTTP požadavku. Django naštěstí poskytuje atribut `method` u vstupního parametru `request`. Tento atribut obsahuje řetězec, psaný velkými písmeny, identifikující danou HTTP metodu. V našem kódu lze tedy pomocí jednoduchých podmínek `if request.method == 'DELETE|PUT|GET|POST'` větvit část programu a reagovat tak na požadované akce.

Dalším krokem, je zpracování požadavku a vrácení odpovědi. Python poskytuje modul `json`, který pomocí metody `json.load()` deserializuje textovou reprezentaci JSON formátu na Python objekt. Obdobně pracuje metoda `json.dumps()`, která naopak serializuje Python objekt na řetězec ve formátu JSON. Pro vrácení odpovědi využijeme funkci `JsonResponse`, dostupnou v modulu `django.http`. Tato funkce bere jako první parametr slovníkový objekt obsahující výsledná data. Na základě těchto znalostí jsme schopni zpracovat takřka jakýkoliv požadavek. Jednoduchá implementace zpracování požadavku pro vytvoření nové události by mohla vypadat následovně:

```
1  if request.method == 'POST':
2    body = json.loads(request.body)
3
4    # Tvorba nove udalosti
5    e = Event()
6    e.title = body['title']
7    e.description = body['description']
8    e.content = body['content']
9    e.start_date = body['startDate']
10   e.end_date = body['endDate']
```

```
11
12    # Prideleni mapy
13    if body['mapId']:
14      e.map = Map.objects.get(id=body['mapId'])
15    else:
16      e.map = None
17
18    # Ulozeni do databaze
19    e.save()
20
21    # Vraceni odpovedi
22    return JsonResponse({
23      'success': True,
24      'data': e.to_json()
25    })
```

Výpis 9.3: Ukázka implementace view pro správu událostí

9.2 Klientská část

V kapitolách 5 a 6 jsem se zmínil o základních konceptech knihovny React a Redux. Opět se budu snažit vysvětlit, vzhledem k rozsáhlosti JavaScriptového kódu, na jednoduchých příkladech procesy spojené s tvorbou komponent a jejich propojení s knihovnou Redux. U ostatních komponent použitých v naší aplikaci, jsou tyto procesy obdobné a není proto třeba se věnovat každé komponentě zvlášť.

9.2.1 Vytváření komponent

V předchozích kapitolách jsem se zmínil o procesu vytváření jednoduchých komponent v knihovně React. V následujících příkladech budeme pokračovat tvorbou komponent pro stránku zabývající se správou událostí a ukážeme si jak vytvořit komponentu pro vykreslení tabulky obsahující jednotlivé záznamy.

Při tvorbě komponent je důležité dokázat rozdělit řešený problém na menší části (komponenty). Na obrázku 9.1 je graficky znázorněna možná dekompozice tabulky, pro zobrazení seznamu událostí, do tří nezávislých komponent.

Obrázek 9.1: Dekompozice tabulky na jednodušší komponenty

Pokud budeme uvažovat o tvorbě tabulky můžeme definovat následující komponenty:

- **`<Events />`** - Jedná se o jakýsi kontejner obsahující pouze danou tabulku. V následujících kapitolách bude tato komponenta použita pro propojení knihovny Redux a vykreslení aplikace do DOMu.
- **`<EventsTable />`** - Obsahuje syntaxi pro vykreslení tabulky v HTML a stará se o zpracování jednotlivých záznamů v podobě řádků tabulky.
- **`<EventsTableRow />`** - Představuje jeden řádek tabulky, zobrazující data konkrétního záznamu.

Pokud předpokládáme, že v proměnné `this.props.events` se nachází pole JSON objektů, obsahujících informace o jednotlivých událostech, implementace výše zmíněných komponent by mohla vypadat následovně.

```
class Events extends React.Component {
  render() {
    return (
      <div>
        <EventsTable />
      </div>
    );
  }
}
```

Výpis 9.4: Implementace komponenty `Events`

```
class EventsTable extends React.Component {
  render() {
    return (
      <div>
        <table>
          <thead>
            <tr>
              <th>Nazev</th>
              <th>Zacatek</th>
              <th>Konec</th>
              <th>Akce</th>
            </tr>
          </thead>
          <tbody>
            {this.props.events.map((event) => {
              return <EventsTableRow key={event.id} event={event} />
            })}
          </tbody>
        </table>
      <div>
```

```
    );
  }
}
```

Výpis 9.5: Implementace komponenty `EventsTable`

Jak již bylo dříve zmíněno, v syntaxi JSX je možné používat výrazy normálního JavaScriptu. V tomto případě 9.5 jsme použili funkci `map` pro průchod polem a vrácení nové komponenty, které zasíláme data daného záznamu.

```
class EventsTableRow extends React.Component {
  render() {
    return (
      <tr>
        <td><strong>{props.event.title}</strong></td>
        <td>{props.event.startDate}</td>
        <td>{props.event.endDate || ""}</td>
        <td><Button>Smazat Udalost</Button></td>
      </tr>
    );
  }
}
```

Výpis 9.6: Implementace komponenty `EventsTableRow`

Jak je vidět na příkladu 9.6 komponenta `EventsTableRow` pouze zobrazuje data, která nám byla předána z rodičovské komponenty `EventsTable`. Přístup k těmto datům máme prostřednictvím objektu `this.props`. Navíc pro atribut `endDate` existuje podmínka, která v případě nulové hodnoty vrátí prázdný řetězec.

9.2.2 Akce

Abychom mohli výše zmíněné komponenty naplnit daty, je třeba tato data nejdříve získat ze serveru. K tomu využijeme definici akcí, které nám následně naplní reducer daty ze serveru. Definice těchto akcí využívá prvků knihovny Redux a nejedná se tedy o API dostupné v Reactu. Jejich syntaxe se proto bude lehce lišit od výše uvedených příkladů pro vytváření komponent.

Kapitola 6.2 blíže popisuje činnost těchto akcí, které vždy vrací objekt s povinným atributem `type`. Proces vytváření akcí, které zpracovávají data asynchronně je složitější než je tomu v případě běžných akcí. Pro vytváření HTTP požadavků a jejich následné zasílání na server bude využito `fetch` API.

Při volání asynchronních akcí existují dva klíčové okamžiky v čase: okamžiky kdy danou akci spustíme a okamžiky kdy získáme odpověď. Každý z těchto okamžiku často vyžaduje změnu stavu aplikace a je proto nutné definovat akce pro každou změnu tohoto stavu. V případě implementace akce pro získání seznamu událostí, definujeme následující tři stavy:

- **načítání** - Okamžik kdy zavoláme akci pro získání dat. V tomto okamžiku je nutné dát aplikaci najevo, že provádíme danou akci. To je možné například změnou hodnoty proměnné `loading` na true, která zobrazí na webové stránce indikátor načítání.
- **chyba** - Okamžik kdy jsme ze serveru získali neplatnou odpověď, nebo nastala chyba v dotazu. V tomto případě je nutné dát aplikaci vědět, že nastala chyba a zároveň změnit hodnotu proměnné `loading` na false pro skrytí indikátoru načítání.
- **úspěch** - Okamžik kdy jsme ze serveru získali platnou odpověď obsahující námi požadovaná data. Získaná data je nutné vložit do reduceru a znovu zajistit skrytí indikátoru načítání změnou hodnoty proměnné `loading`.

Na základě těchto znalostí můžeme identifikovat tři akce a jednu funkci, která obstarává vytváření HTTP požadavku:

- **`fetchEventsRequest()`** - Akce, volána v okamžiku vytvoření HTTP požadavku, která zajistí změnu hodnoty proměnné `loading` na `true`.
- **`fetchEventsError(error)`** - Akce, volána v případě, že nastane chyba. Předává obsah chyby reduceru prostřednictvím parametru `error` a zajišťuje změnu hodnoty proměnné `loading` na `false`.
- **`fetchEventsSuccess(data)`** - Akce, volána v případě plané odpovědi ze serveru. Předává data získaná ze serveru reduceru a podobně jako předešlá funkce zajišťuje změnu hodnoty proměnné `loading` na `false`.
- **`fetchEvents()`** - Funkce, která vytváří HTTP požadavek prostřednictvím `fetch` API a v závislosti na typu odpovědi volá dané akce.

V závislosti na výše definovaných okolnostech bude implementace těchto funkcí vypadat následovně:

```
1  export const FETCH_EVENTS_REQUEST = 'FETCH_EVENTS_REQUEST';
2  export const FETCH_EVENTS_ERROR = 'FETCH_EVENTS_ERROR';
3  export const FETCH_EVENTS_SUCCESS = 'FETCH_EVENTS_SUCCESS';
4
5  export function fetchEventsRequest() {
6    return {
7      type: FETCH_EVENTS_REQUEST
8    };
9  }
10
11 export function fetchEventsError(error) {
12   return {
13     type: FETCH_EVENTS_ERROR,
14     error: error
```

```
15    };
16  }
17
18  export function fetchEventsSuccess(data) {
19    return {
20      type: FETCH_EVENTS_SUCCESS,
21      events: data
22    };
23  }
24
25  export function fetchEvents() {
26    return dispatch => {
27      dispatch(fetchEventsRequest());
28
29      fetch('/events/'), { method: 'GET' })
30        .then(response => {
31          return response.json();
32        })
33        .then(data => {
34          dispatch(fetchEventsSuccess(data));
35        })
36        .catch(error => {
37          dispatch(fetchEventsError(error))
38        });
39    };
40  }
```

Výpis 9.7: Implementace akcí pro získání seznamu událostí ze serveru

Můžeme si všimnout, že funkce `fetchEvents` vrací novou funkci s parametrem `dispatch`. Jedná se o využití modulu `redux-thunk`, který přidává funkčnost vytváření asynchronních akcí, jenž knihovna Redux implicitně nepodporuje.

9.2.3 Reducer

Po definici akcí je nutné také vytvořit `reducer`, který nám říká jak dané akce mění stav aplikace. Více o reduceru je zmíněno v kapitole 6.3. Při vytváření reducerů je nejdříve nutné definovat výchozí stav aplikace. V našem případě bude aplikace obsahovat pole událostí, proměnnou `loading` a řetězcovou proměnou `error`. Ta bude uchovávat záznam poslední chyby, která nastala v rámci volání akcí. Implementace výchozího stavu aplikace vypadá následovně:

```
1  const inititalState = {
2    events: [],
3    loading: false,
4    error: ''
5  };
```

Výpis 9.8: Výchozí stav aplikace pro správu událostí

Dalším krokem je vytvoření samotného reduceru, ten je funkcí, která má dva parametry: stav (`state`) a akce (`action`). Uvnitř tohoto reduceru musíme zajistit správné chování v závislosti na typu volané akce. Pokud je nám typ akce neznámý, vracíme vždy původní stav aplikace. Reducery v případě změny vrací vždy nový objekt, to výrazně zrychluje zpracování jednotlivých akcí, ale také přispívá k jednoduchému testování. Vytváření nového stavu v závislosti na přijatých datech, lze docílit velice jednoduše pomocí operátoru restrukturalizace `...`, dostupného při použití syntaxe ECMAScript 2015.

Implementace tohoto reduceru s využitím konstanty `inititalState`, definované v předchozí ukázce, a objektu `actions`, obsahujícím všechny typy dříve vytvořených akcí, bude tedy vypadat následovně:

```
1  const events = (state = inititalState, action) => {
2    switch (action.type) {
3      case actions.FETCH_EVENTS_SUCCESS:
4        return {
5          ...state,
6          loading: false,
7          events: [
8            ...action.events
9          ]
10     };
11
12     case actions.FETCH_EVENTS_ERROR:
13       return {
14         ...state,
15         error: action.error,
16         loading: false
17       };
18
19     case actions.FETCH_EVENTS_REQUEST:
20       return {
21         ...state,
22         loading: true
23       };
24
25     default:
26       return state;
27   }
28 };
```

Výpis 9.9: Implementace reduceru pro správu událostí

9.2.4 Vytvoření Redux store

Posledním krokem je propojení akcí a reducerů s komponentami v Reactu. K tomu využijeme rozšíření `react-redux`, které poskytuje metodu `connect` pro připojení jednotlivých kompo-

nent k Redux store a komponentu `Provider`, která slouží k přiřazení store naší aplikaci. Použití této komponenty spočívá v obalení rodičovské komponenty (v našem případě komponenta `EventsTable` uvnitř komponenty `Events`) a přiřazení daného store prostřednictvím stejnojmenného argumentu.

Nejdříve však musíme vytvořit store z námi předem vytvořených reducerů a zároveň také aplikovat middleware pro zajištění podpory asynchronních akcí. K tomu využijeme funkce `createStore()`, `applyMiddleware()` a objektu `thunk`, dostupného z modulu `redux-thunk`. Použití těchto funkcí vypadá následovně:

```
1 const createStoreWithMiddleware = applyMiddleware(thunk)(createStore);
2 const store = createStoreWithMiddleware(events);
```

Výpis 9.10: Vytvoření store s aplikací thunk middleware

Funkce `applyMiddleware()` vrací novou funkci `createStoreWithMiddleware()`. Ta bere jako vstupní parametr námi vytvořený reducer, v tomto případě `events`. Funkce `applyMiddleware()` zajišťuje, že všechny volané akce v rámci vytvořeného store, budou proudit přes modul `thunk`, který zajišťuje podporu asynchronních akcí. Nakonec už jen zbývá „obalit“ komponentu `EventsTable` uvnitř komponenty `Events` komponentou `Provider` a přiřadit ji vytvořený store.

```
1 class Events extends React.Component {
2   render() {
3     return (
4       <Provider store={store}>
5         <EventsTable />
6       </Provider>
7     );
8   }
9 }
```

Výpis 9.11: Propojení aplikace s Redux Store

9.2.5 Propojení komponent s Redux store

V předchozí kapitole jsem se zmínil o funkci `connect` dostupné v rozšíření `react-redux`, která slouží k namapování dat v Redux store na vlastnosti dané komponenty. Tato funkce zároveň slouží k vázání akcí na funkci `dispatch()` a jejich následné namapování na vlastnosti komponenty, které umožní jejich volání bez nutnosti volání funkce `dispatch()`.

Funkce `connect` bere dva argumenty. Prvním argumentem je funkce s parametrem `state` vracející objekt namapovaných vlastností. Druhým argumentem je funkce s parametrem `dispatch`, která slouží pro vázání akcí na funkci `dispatch()` s využitím pomocné funkce `bindActionCreators()`. Funkce se používá v části zdrojového, kódu kdy exportujeme třídu dané komponenty a její použití vypadá následovně:

```
export default connect(
  (state) => {
    return {
      events: state.events.events,
      loading: state.events.loading,
      error: state.events.error
    };
  }, (dispatch) => {
    return bindActionCreators({ fetchEvents }, dispatch);
  }
)(EventsTable)
```

Výpis 9.12: Mapování dat z Redux store na vlastnosti komponenty

Po provedení těchto změn máme v komponentě `EventsTable` k dispozici data z námi dříve vytvořeného reduceru a akci pro získání seznamu událostí ze serveru prostřednictvím objektu `this.props`.

9.2.6 Připojení komponenty k DOMu

Posledním krokem k vytvoření funkčního uživatelského rozhraní s využitím knihovny React, je připojení rodičovské komponenty k DOMu. To zle provést pomocí metody `ReactDOM.render()` obsažené v modulu `react-dom`. Prvním argumentem této metody je název komponenty, kterou chceme připojit k DOMu a druhým je uzel stromu, jehož potomkem by se měla komponenta stát. Tento uzel může být konkrétní prvek s daným ID, který lze najít s využitím funkce `document.getElementById()` nebo jakýkoliv jiný uzel DOM stromu. V případě naší ukázky, připojíme komponentu k předem vytvořenému HTML elementu s ID `react-root`:

```
ReactDOM.render(<Events />, document.getElementById('react-root'));
```

Výpis 9.13: Mapování dat z Redux store na vlastnosti komponenty

9.3 Shrnutí

V předchozích několika kapitolách jsem se snažil přiblížit implementaci webové aplikace s využitím aplikačního frameworku Django na straně serveru a knihoven React a Redux pro uživatelské rozhraní klientské části. Vzhledem k rozsahu aplikace nebylo možné pokrýt všechny její části, a proto jsem se spíše soustředil na vysvětlení procesů použitých při vývoji na jednoduchých příkladech. Tato aplikace používá v částech, které zde nebyly popsány, obdobné, ne-li stejné, techniky a není proto nutné je zde dále popisovat.

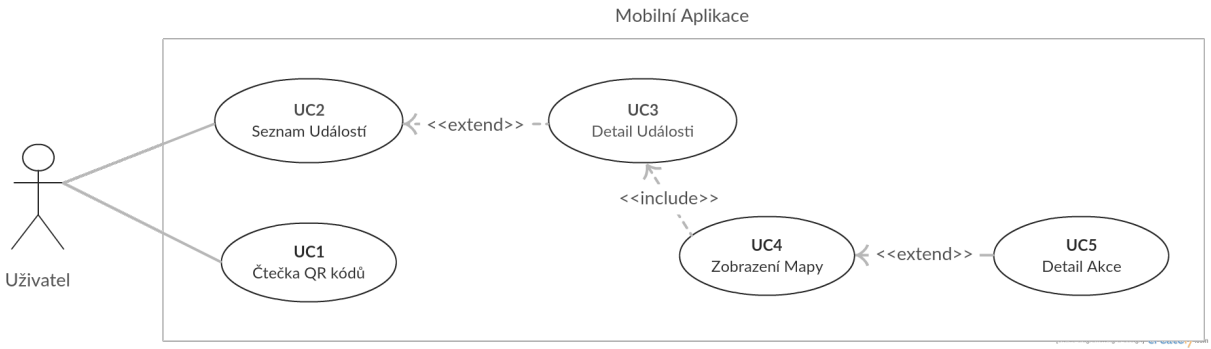
10 Návrh mobilní aplikace

Výsledkem této bakalářské práce by měla být, mimo implementaci webové aplikace, také plně funkční mobilní aplikace dostupná pro zařízení s operačním systémem Android a iOS, která by měla sloužit výhradně pro zobrazení dat vytvořených webovou aplikací a uchovávání průběhu a informací o probíhajících hrách (soutěžích).

10.1 Specifikace zadání

Před specifikací požadavků a návrhem uživatelského rozhraní mobilní aplikace je nutné blíže určit zadání požadovaných funkcí. Po konzultaci s vedoucím práce bylo zadání pro mobilní aplikaci specifikováno následujícími několika body:

1. Mobilní aplikace bude zobrazovat seznam událostí získaných z webového API, po jejichž kliknutí zobrazí nové okno s definovaným obsahem.
2. Mobilní aplikace bude zobrazovat mapy přiřazené jednotlivým událostem.
3. Mapy mobilní aplikace by měly podporovat možnost vytváření akčních objektů, které po kliknutí aplikace zobrazí další informace.
4. Mobilní aplikace by měla zobrazovat dostupné hry (soutěže) jednotlivých událostí a ukládat jejich postup v podobě čtení speciálních QR kódů.
5. Mobilní aplikace bude implementována s využitím JavaScriptového frameworku React Native.



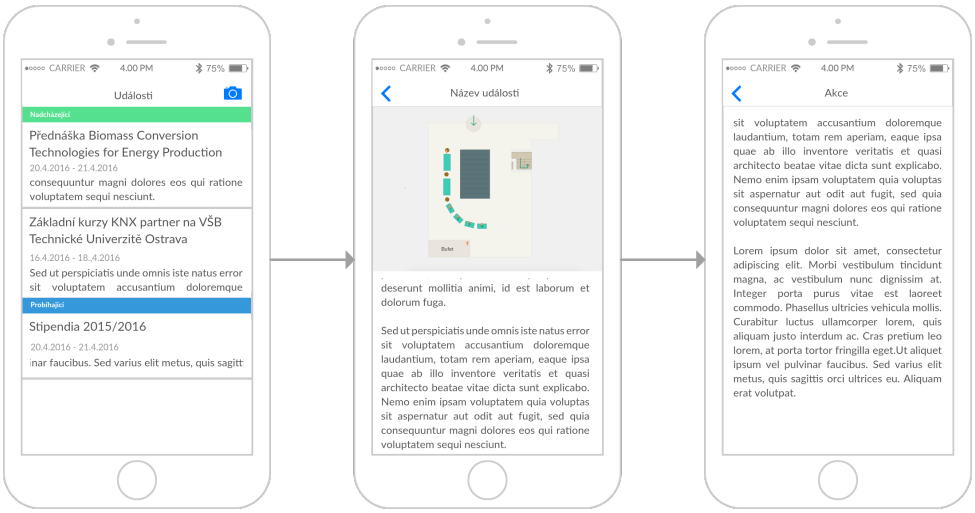
Obrázek 10.1: Diagram případů užití v mobilní aplikaci

10.2 Návrh uživatelského rozhraní

Proces návrhu uživatelských rozhraní pro mobilní zařízení je jiný, než je tomu v případě webových stránek. Při jeho návrhu musíme dbát na omezení těchto zařízení, mezi které patří např.: menší velikost displeje, jiná forma zpracování vstupů a další.

K řešení těchto problémů se často využívají prvky seznamů a navigace, které nám umožní definovat strukturu pohledů, podobnou principu datové struktury zásobníků známe z programovacích jazyků, kdy můžeme jednotlivé pohledy měnit pomocí metod `pop` a `push`. S využitím těchto prvků lze velice jednoduše vytvořit aplikaci, která obsahuje více pohledů, umožňuje tedy zobrazit neomezené množství informací a zároveň poskytnout jednoduchou navigaci mezi nimi.

Vzhledem k jednoduchosti mobilní aplikace, která neobsahuje žádné složité vstupní pole, ale slouží především pro korektní zobrazení dat vytvořených webovou aplikací, můžeme definovat čtyři hlavní okna.



Obrázek 10.2: Návrh uživatelského rozhraní mobilní aplikace

10.2.1 Seznam událostí

Jedná se o první okno, které uživatel uvidí po spuštění aplikace. Obsahuje seznam událostí seřazený do tří skupin: nadcházející, probíhající a ukončené. Jednotlivé položky seznamu zobrazují titulek, datum trvání a krátký popis události. Po kliknutí na jakoukoliv položku seznamu se zobrazí nové okno s detailem události. Pro aktualizaci seznamu je zde k dispozici další ovládací prvek, často využívaný u jiných mobilních aplikací, `PullToRefresh`. Jedná se o možnost aktualizace seznamu událostí jeho stažením dolů, dokud se uživateli neukáže indikátor zobrazující průběh aktualizace. Nakonec se v pravém horním rohu nachází tlačítko pro zobrazení modálního okna čtečky QR kódů.

10.2.2 Detail události

Hlavním prvkem tohoto okna je plně interaktivní mapa vytvořená editorem mapy prostřednictvím webové aplikace. Dále jsou zde zobrazeny detailní informace o zvolené události a právě probíhající soutěži.

10.2.3 Detail akce

Akcí rozumíme obsah přiřazený jednotlivým objektům na mapě, toto okno se tedy zobrazí po kliknutí na objekt, jemuž je přiřazená nějaká akce.

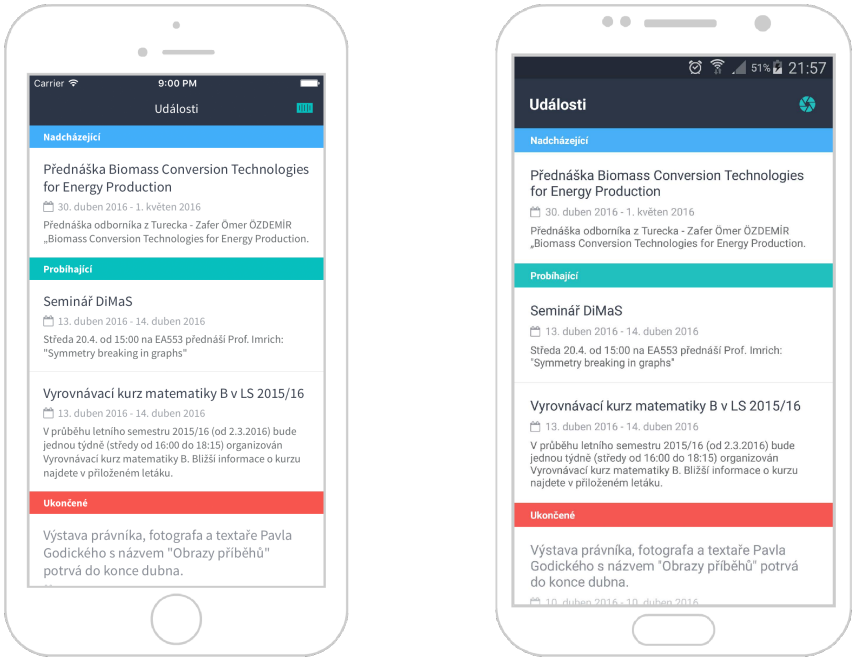
10.2.4 Čtečka QR Kódů

Toto modální okno můžeme zobrazit po kliknutí na příslušné tlačítko v seznamu událostí. Hlavním prvkem je zobrazení pohledu kamery, který podporuje čtení QR kódů vytvořených webovou aplikací.

10.3 Design uživatelského rozhraní

Design uživatelského rozhraní mobilní aplikace by měl klást důraz na jednoduchost a přehlednost. Vzhledem k omezení velikosti displeje mobilních zařízení je třeba dbát na zobrazení pouze důležitých a relevantních informací. Zároveň by měl design odpovídat pravidlům definovaným pro jednotlivé platformy.

Vzhledem k tomu, že vytváříme mobilní aplikaci, která poběží na dvou platformách, je nutné zajistit, aby všechny ovládací prvky aplikace zapadaly do daného ekosystému. To lze zajistit s využitím obecných komponenty (`<Navigator />`, `<RefreshControl />`), které poskytují vzhled odpovídající nativním komponentám dané platformy. Barevné schéma mobilní aplikace vychází s dříve definovaného designu webové stránky z důvodů zajištění konzistence. Rozložení jednotlivých prvků odpovídá návrhu a je dáno použitím nativních komponent, které zajistí jejich korektní vykreslení.



Obrázek 10.3: Srovnání designu mobilní aplikace na platformách iOS (vlevo) a Android (vpravo)

11 Implementace mobilní aplikace

Jelikož framework React Native vychází z JavaScriptové knihovny React a mimo dodatečných API určených pro přístup k nativním modulům mobilních platforem, je proces vytváření komponent takřka identický. Nebudu se tedy, jako tomu bylo v předchozí kapitole, věnovat popisu vytváření jednoduchých komponent a jejich propojení s knihovnou Redux, ale spíše zde popíšu řešení dvou hlavních problémů, které vznikly při vývoji této aplikace.

Tyto problémy zároveň reprezentují hlavní prvky mobilní aplikace, kterými jsou vykreslení mapy a schopnost reagovat na kliknutí na jednotlivé objekty na mapě a v závislosti na nich zobrazit nové okno s dalšími informacemi. Tento, ač na první pohled jednoduchý úkol, se ukázal být díky omezením této knihovny relativně složitý.

11.1 API poskytované webovou aplikací

Před popisem řešení výše zmíněných problémů je nutné se zmínit o API poskytovaných webovou aplikací sloužících pro přístup k vytvořeným datům a formátu získávaných dat. Webové API využívá architektury REST, o které jsem se již zmínil v kapitole 9.1.1. Jelikož mobilní aplikace slouží výhradně k zobrazení vytvořených dat, budeme používat především HTTP metodu `GET` k jejich získání. Využití architektury REST zároveň také znamená, že každý záznam je jednoznačně identifikovatelný svým URL. Mezi URL definované webovým API patří:

- **`/api/v1/events`** - Výpis všech aktuálních událostí.
- **`/api/v1/events/<ID>`** - Detailní informace konkrétní události s daným ID.
- **`/api/v1/maps`** - Výpis všech map.
- **`/api/v1/maps/<ID>`** - Data konkrétní mapy s daným ID.

Všechny výše uvedené URL obsahují prefix **`/v1`**, tento prefix je zde z důvodu zachování jednoduché rozšiřitelnosti těchto API. V případě, že se v budoucnu rozhodneme webovou aplikaci rozšířit o nové funkce, které budou vyžadovat změnu formátu přenášených dat, můžeme velice jednoduše vytvořit nové API s novým prefixem **`/v2`** a zároveň tak zachovat podporu i pro starší API, které mohou být využívány současnými aplikacemi.

11.1.1 Formát dat

Všechna data vrácená webovým API jsou ve formátu JSON. To umožňuje jednoduché zpracování v React Native bez nutnosti dalších konverzí. V rámci webového API máme k dispozici dvě entity: mapa a událost. Ty mají následující formát:

```
1 var udalost = {
2   "startDate": "2016-03-13T22:06:35.468Z",
3   "endDate": "2016-04-13T22:06:35.468Z",
```

```
4     "title": "Nova udalost",
5     "description": "",
6     "mapId": 89,
7     "content": "",
8     "game" null,
9     "id": 117
10  }
11
12  var mapa = {
13    "updated": null,
14    "mapData": {
15      "styles": {},
16      "svg: []
17    },
18    "created": "2016-03-04T09:20:36.997Z",
19    "title": "Nova Mapa",
20    "id": 89,
21    "description": ""
22  }
```

Výpis 11.1: Datový formát entit v rámci webového API

U události se nachází objekt `content`, jedná se o obsah, který podporuje formátovaný text, uložený ve formátu HTML. Tento obsah je ve webové aplikaci tvořen editorem Draft.js[7], který umožňuje serializaci formátovaného obsahu do JSON objektu a jeho následný převod na syntaxi HTML. HTML obsah je poté nutné v rámci mobilní aplikace zpracovat a zajistit jeho korektní zobrazení. Není tedy možné jej vypsat přímo. K zpracování tohoto obsahu se využívá vlastní komponenty `<RichContent />`. Dalším významným atributem je objekt `game`, který obsahuje další informace o probíhající soutěži přiřazené dané události.

Mapa obsahuje atribut `mapData`, ten dále globální definici stylů pro danou mapu `style` a objektovou reprezentaci svg elementů na mapě `svg`. Více o zpracování tohoto formátu a atributech jednotlivých svg objektů se zmíním v následující kapitole.

11.2 Implementace vykreslení mapy

Jedním z hlavních úkolů při návrhu této aplikace bylo určit, jaký datový formát bude použit pro přenos mapy mezi webovou a mobilní aplikací a jakým způsobem bude v mobilním zařízení zobrazena. Téměř okamžitě byl po konzultaci s vedoucím práce určen pro přenos a zobrazení mapy formát SVG, o kterém jsem se více zmínil v kapitole 2.

Tento formát nabízí syntaxi pro tvorbu dvourozměrné vektorové grafiky uvnitř webových stránek, navíc je v dnešní době podporován ze strany všech hlavních webových prohlížečů. Díky těmto vlastnostem se tak stal vhodným kandidátem pro řešení toho problému.

[7] Draft.js - knihovna pro tvorbu textových editorů s podporou formátování psaná v Reactu.

11.2.1 Zápis SVG pomocí JSONu

Zápis dat v SVG, neumožňuje používání vlastních značek pro zajištění další funkcionality mapy jako je např. definice akčních objektů. Zároveň je práce s XML v JavaScriptu výrazně obtížnější než je tomu v případě použití JSONu. Bylo proto nutné přijít s vlastním formátem, který reprezentuje jednotlivé SVG elementy, podporuje zápis vlastních dat, a přitom umožňuje jednoduchou manipulaci v JavaScriptu.

Výsledkem tohoto snažení byl zápis JSON objektu, který reprezentuje jednotlivé SVG elementy a přitom podporuje ukládání vlastních dat. Použití formátu JSON zároveň ulehčuje práci s daty v JavaScriptu, kdy je možné s ním pracovat přímo. Výsledný formát popisující svg objekty obsahuje následující vlastnosti:

- **`attr`** - Objekt představující vlastnosti SVG elementu a jejich hodnoty.
- **`data`** - Různá data používaná pro zajištění další funkcionality, např. u elementu `text` se jedná o obsah, který má být zobrazen.
- **`type`** - Řetězec reprezentující typ SVG elementu, aplikace podporuje elementy `path`, `rect`, `circle`, `ellipse`, `text` a `line`.
- **`action`** - Titulek a obsah akce, přiřazené danému SVG elementu.
- **`style`** - Vlastní styly daného elementu zapsané pomocí JSON objektu.

```
{
  type: 'rect',
  action: {},
  style: {
    fill: 'red'
  },
  data: {},
  attr: {
    width: 0,
    height: 0,
    x: x,
    y: y
  }
}
```

Výpis 11.2: Zápis SVG elementu pomocí JSON objektu

Výše uvedený příklad zároveň popisuje formát zápisu jednotlivých SVG objektů, které se nachází v atributu `mapData` získaného z webového API popsaného v kapitole 11.1.1. Seznam těchto objektů je pak velice jednoduché zpracovat a vykreslit.

11.2.2 Řešení problému vykreslení mapy

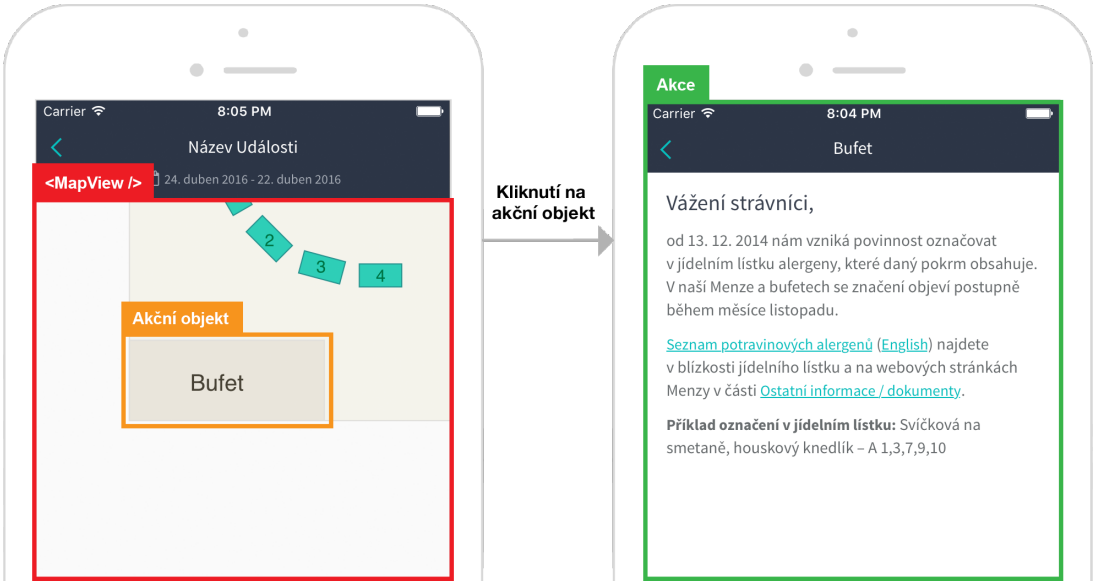
Po vyřešení problémů spojených s přenosem a formátem dat bylo nutné mapu vykreslit. Prvním pokusem tohoto snažení bylo použití externí knihovny `react-native-svg`. Ta zpřístupňuje `SVGKit` API, které umožňuje vykreslit nativní SVG vektorovou grafiku. Použití této knihovny pro více než jednoduché tvary se však ukázalo být složité. Navíc nepodporovala události kliknutí na jednotlivé objekty, podpora ze strany vývojáře byla téměř nulová, a její použití bylo omezeno na platformu iOS.

Dalším pokusem bylo vygenerování HTML stránky obsahující danou mapu v SVG a její následné vykreslení prostřednictvím komponenty `<WebView />`. Výhodou tohoto řešení byla nezávislost na externích knihovnách pro vykreslení SVG a ve srovnání s předchozím řešením, kdy jsme použili knihovnu `react-native-svg`, zlepšení výkonu vykreslování. Navíc, vzhledem k tomu, že se jedná o webovou stránku, komponenta implicitně podporuje gesta pinch-to-zoom (v případě platformy iOS) pro přiblížení a plynulý posun mapy. V naší aplikaci toto vykreslování zprostředkovává vlastní komponenta `<MapView />`, kterou si popíšeme v následující kapitole.

11.2.3 Komponenta <MapView />

Jak již bylo řečeno v předchozí kapitole, hlavním prvkem komponenty `<MapView />` je komponenta `<WebView />`, která je implicitně podporována knihovnou React Native. Ta se stará o vykreslení vygenerované webové stránky obsahující danou mapu.

Povinným parametrem komponenty `<MapView />` je `source`, kterému předáváme objekt `mapData` získaný z webového API. Mezi další důležité složky této komponenty můžeme zařadit několik funkcí, které se starají o správný převod objektové reprezentace SVG elementů na syntaxi pro použití v HTML. Popis těchto funkcí je vyjádřen v několika následujících kapitolách. Pro jednodušší orientaci v popisu a implementaci jednotlivých komponent níže uvedený obrázek popisuje výsledný produkt.



Obrázek 11.1: Výsledné použití komponenty `<MapView />`

Funkce parseStyles(styles)

Tato funkce se stará jednak o vytvoření textového řetězce atributu style z obsahu objektu `style`, ale také o přidání nutných vendor prefixů v případě využití vlastnosti transform, která je využita pro rotaci SVG elementů na mapě. Názvy jednotlivých vlastností stylů jsou psány pomocí konvence camelCase a je proto nutné nahradit velká písmena na začátku slov malými a „mezery“ mezi slovy pomlčkou, abychom mohli názvy těchto vlastností použít pro styly CSS. Implementace této funkce vypadá následovně:

```
1  parseStyles(styles) {
2    var parsedStyles = 'style="';
3    var property = '';
4
5    // Prevod objektu style na retezec
6    for (var prop in styles) {
7      if (styles.hasOwnProperty(prop)) {
8        // Prevod camelCase nazvu vlastnosti na dash-case
9        property = prop.replace(/\.?([A-Z]+)/g, function (x,y) {
10         return '-' + y.toLowerCase()
11       }).replace(/^_/, '');
12
13       // Vlozeni vendor webkit prefixu
14       if (prop.match(/transform/)) {
15         parsedStyles += `-webkit-${property}: ${styles[prop]}; `;
16       }
17       parsedStyles += `${property}: ${styles[prop]}; `;
18     }
19   }
```

```
20
21    // Odstraneni posledni prazdne mezery a pridani uvozovky
22    return parsedStyles.substr(0, parsedStyles.length - 1) + "\"";
23  }
```

Výpis 11.3: Implementace funkce pro převod stylů

Funkce parseAttributes(attr)

Dalším krokem je parsování atributů objektu `attr`. Jelikož názvy všech atributů odpovídají jejich názvu v SVG, není nutné provádět jakékoliv konverze a stačí pouze z jednotlivých atributů vytvořit řetězce odpovídající jejich syntaxi v SVG. Její implementace vypadá následovně:

```
1   parseAttributes(attr) {
2     var parsedAttr = '';
3
4     // Prevod nazvu vlastnosti na retezec
5     for (var prop in attr) {
6       if (attr.hasOwnProperty(prop)) {
7         parsedAttr += `${prop}="${attr[prop]}" `;
8       }
9     }
10
11    // Odstraneni posledni prazdne mezery
12    return parsedAttr.substr(0, parsedAttr.length - 1);
13  }
```

Výpis 11.4: Implementace funkce pro zpracování atributů

Funkce parseSvgElement(svg)

Tato funkce se stará o vykreslení korektního elementu SVG v závislosti na jeho typu. Zároveň v případě elementu `text` vypíše obsah definovaný v atributu `svg.data.content`. Při tvorbě SVG elementů tato funkce volá funkce pro parsování atributů a stylů a vrací řetězec obsahující validní SVG element. Implementace této funkce je zobrazena níže:

```
1   parseSvgElement(svg) {
2     if (svg.type == 'text') {
3       return `<text ${this.parseStyles(svg.style)} ${this.parseAttributes(svg.attr)}>${svg.data.
        content}</text>`;
4     } else {
5       return `<${svg.type} ${this.parseStyles(svg.style)} ${this.parseAttributes(svg.attr)} />`;
6     }
7   }
```

Výpis 11.5: Implementace funkce pro zpracování SVG elementů

Funkce getMap(source)

Jedná se o poslední důležitou složku v rámci zpracování mapy. Tato funkce prochází jednotlivé objekty SVG získané z webového API, pro které volá výše zmíněnou funkci `parseSvgElement(svg)`. Ta vrací řetězec reprezentující validní SVG element. Výsledkem volání této funkce je řetězec zpracovaných elementů vložený do předem připraveného řetězce s HTML syntaxí pro vytvoření jednoduché webové stránky. Ta je následně použita jako zdroj pro komponentu `<WebView />`, který danou mapu vykreslí na obrazovku zařízení. Implementace tohoto řešení vypadá následovně:

```
1  getMap(source) {
2    if (source == null) {
3      return '';
4    }
5
6    var parsedSvg = source.svg.map(svg => {
7      return this.parseSvgElement(svg);
8    }).join('');
9
10   return `<!doctype html><html style="background: #fafafa;" lang="en"><head><meta charset="UTF-8
       "><title>Map</title></head><body style="padding: 40px; background: #fafafa;"><svg width="
       1088" height="600" xmlns="http://www.w3.org/2000/svg">${parsedSvg}</svg></body></html>`;
11 }
12
13 render() {
14   return <WebView source={{ html: this.getMap(this.props.source) }}/>;
15 }
```

Výpis 11.6: Implementace funkce pro vykreslení mapy a její použití

11.3 Implementace akčních objektů na mapě

V předchozí kapitole jsem se zmínil, jakým způsobem je řešeno vykreslení mapy. Dalším úkolem je schopnost reagovat na kliknutí na jednotlivé objekty na mapě, kterým je přiřazena nějaká akce. Problém s použitím komponenty `<WebView />` je ten, že neposkytuje žádné metody, kterými bychom mohli přímo komunikovat s obsahem webové stránky a nativní částí aplikace. K vyřešení tohoto problému využijeme přiřazení callback funkce na vlastnost `onNavigationStateChange`, která vždy vrací aktuální `url` v případě jeho změny v rámci webové aplikace.

11.3.1 Použití funkce onNavigationStateChange()

Jediná část URL webové stránky, jejíž změna nezpůsobí její znovu načtení, je kotva (`hash`). Tuto kotvu tedy můžeme použít pro přiřazení vlastních dat v rámci webové stránky, vyvolání změny URL a její následné zachycení v nativní částí aplikace pomocí callback funkce přiřazené vlastnosti `onNavigationStateChange()`.

Jelikož přiřazení stejné hodnoty `hash` nevyvolá spuštění této callback funkce (tento problém může nastat, pokud uživatel několikrát klikne na stejnou akci), musíme zajistit to, že hodnota `hash` bude vždy unikátní. To lze velice jednoduše vyřešit aplikováním prefixu generující náhodné číslo, který jsme schopni bezpečně odstranit při následné extrakci dat z URL.

Prvním úkolem je tedy registrování a zachycení událostí kliknutí na jednotlivé SVG elementy, uvnitř webové stránky, které mají přiřazenou nějakou akci. K zajištění této funkcionality je nezbytné rozšířit funkci `parseSvgElement(svg)` z předchozí kapitoly. Abychom mohli zjistit, která akce je volána, je nutné přiřadit jednotlivým elementům ID, které slouží k její identifikaci. Jelikož standart SVG 1.1 nepodporuje používání vlastních `data-*` atributů dostupných v rámci HTML5 pro přiřazení aplikačně specifických vlastností daným elementům, využijeme dostupného elementu `<g />`, který slouží k seskupení jiných SVG elementů. Tomuto elementu přiřadíme atribut `id`, který bude identifikovat danou akci. Implementace rozšíření funkce z předchozí kapitoly vypadá následovně:

```
1  parseSvgElement(svg) {
2    if (svg.type == 'text') {
3      return `<g ${this.parseAction(svg.action)}><text ${this.parseStyles(svg.style)} ${this.
       parseAttributes(svg.attr)}>${svg.data.content}</text></g>`;
4    } else {
5      return `<g ${this.parseAction(svg.action)}><${svg.type} ${this.parseStyles(svg.style)} ${
       this.parseAttributes(svg.attr)} /></g>`;
6    }
7  }
```

Výpis 11.7: Implementace funkce pro zpracování akcí

Z výše uvedeného příkladu si lze všimnout, že jsme použili funkci `parseAction(action)`, která ještě není definována. Tato funkce nám vrací řetězec s atributem `id` a unikátním číselným ID, které spravuje vnitřní proměnná `actionIdsCounter`. Zároveň ukládá jednotlivé akce do asociativního pole uvnitř třídní proměnné pro jejich jednoduchou pozdější identifikaci. Implementace této funkce je zobrazena níže:

```
1  parseAction(action) {
2    // Kontrola zda akce existuje
3    if (!action.hasOwnProperty('content')) {
4      return '';
5    }
6
7    // Vlozeni akce do promenne a navyseni pocitadla
8    this.actions[this.actionIdsCounter] = action;
9    this.actionIdsCounter++;
10
11   return `id="${this.actionIdsCounter - 1}"`;
12 }
```

Výpis 11.8: Rozšíření funkce pro zpracování SVG elementů v komponentě `<MapView />`

Posledním krokem je registrace událostí kliknutí na SVG elementy s přiřazenou akcí. Tento problém řeší jednoduchý skript, který je vložen do webové stránky s využitím vlastnosti `injectedJavaScript`. Ten registruje události kliknutí na všechny `<g />` elementy webové stránky. Po kliknutí na některý z těchto elementů webová stránka mění obsah `hash` části URL s hodnotu atributu `id`. Toto ID je následně odchyceno registrováním callback funkce na vlastnost `onNavigationStateChange`, kde je následně extrahováno z URL webové stránky. Získaný řetězec je poté převeden na číslo, které slouží pro výběr správné akce z předem vytvořeného asociativního pole. Nakonec je zavolána callback metoda přiřazená vlastnosti `onActionClick` námi vytvářené komponenty `<MapView />` a obsah akce je předán nadřazené komponentě k jejímu dalšímu zpracování. Implementace tohoto řešení vypadá následovně:

```
1  const injectScript = '(function() {
2    var elms = document.querySelectorAll("g");
3    for (var i = 0; i < elms.length; i++) {
4      elms[i].addEventListener("click", function (e) {
5        var actionId = this.getAttribute("id");
6        document.location.hash = ((Math.floor((Math.random() * 10000) + 1)).toString() + "___" +
       actionId).toString();
7      }, false);
8    }
9  })();'
10
11 class MapView extends React.Component {
12   ...
13
14   handleOnStateChanged(navState) {
15     // Extrahovani id akce z~url
16     var parts = navState.url.split('___');
17     var id = parseInt(parts[parts.length - 1]);
18     this.props.onActionClick(this.actions[parseInt(id)]);
19   }
20
21   render() {
22     return <WebViewBridge
23       injectedJavaScript={injectScript}
24       onNavigationStateChange={(navState) => this.handleOnStateChanged(navState)}
25       source={{
26         html: this.getMapHtml(this.props.source)
27       }} />;
28   }
29 }
```

Výpis 11.9: Rozšíření funkce pro vykreslení mapy v komponentě `<MapView />`

11.4 Shrnutí

V předchozích několika kapitolách jsem se snažil představit praktiky vývoje mobilních aplikací s využitím frameworku React Native. Zmínil jsem se o formátu API poskytovaném webovou aplikací a jeho použití v mobilní aplikaci. Zároveň jsem se věnoval popisu konkrétního řešení problému vykreslení mapy v rámci mobilní aplikace. Vzhledem k podobnosti s frameworkem React nebylo nutné se věnovat popisu vytváření jednoduchých komponent, který je takřka identický s postupy zmíněnými v kapitole 9.

12 Závěr

V této bakalářské práci byly představeny technologie použité při tvorbě zadané mobilní a webové aplikace. Mezi ty nejdůležitější můžeme zařadit JavaScriptové knihovny React a Redux určené pro tvorbu uživatelských rozhraní webových stránek, React Native pro tvorbu mobilních aplikací psaných v JavaScriptu a webově aplikační framework Django psaný v jazyce Python.

Na základě specifikovaného zadání byla navrhnuta a implementována webová aplikace pro editaci prezentovaných informací. Tato aplikace pro svou klientskou část využívá JavaScriptové knihovny React a Redux, a komunikuje se serverovou částí prostřednictvím technologie AJAX. To redukuje zbytečné načítání stránek a nabízí tak rychlé a dynamické uživatelské rozhraní pro správu dat. Další součástí webové aplikace je plnohodnotný SVG editor, jenž slouží pro snadné vytváření a editaci map, které jsou důležitou součástí celého konceptu této práce. Aplikace zároveň nabízí veřejné API pro jednoduchý přístup k vytvořeným datům z prostředí mobilní aplikace.

Dalším výsledkem této práce je mobilní aplikace dostupná pro platformy iOS a Android, napsaná s využitím frameworku React Native, která slouží k zobrazení dat, vytvořených prostřednictvím webové aplikace. Mimo zobrazení jednotlivých událostí aplikace nabízí interaktivní mapu, která podporuje události kliknutí na jednotlivé objekty na mapě. Tyto objekty následně zobrazí nové okno s dalšími informaci. Mobilní aplikace také podporuje čtení QR kódů, vygenerovaných webovou aplikací, které nabízí další funkčnost jako zobrazení konkrétní události, čtení soutěžních kódů, či ukázání aktuální polohy na mapě.

Výsledná implementace mobilní aplikace bude nasazena do provozu v čase příležitostí, jako jsou dny otevřených dveří nebo konference a až její chod v rukou uživatelů ukáže, zda je nutné provést nějaké dodatečné úpravy.

Literatura

[1] CHEN, Chun-houh, Wolfgang Härdle a Antony Unwin. Handbook of data visualization. Berlin: Springer, 2008. ISBN 9783540330370.

[2] React Native – A framework for building native apps using React. React Native. [online]. 26.3.2015 [cit. 2016-01-10]. Dostupné z: https://facebook.github.io/react-native/

[3] React – A JavaScript library for building user interfaces. React. [online]. 26.3.2015 [cit. 2015-12-10]. Dostupné z: https://facebook.github.io/react/

[4] ABRAMOV, Dan. Redux. Redux. [online]. 14.8.2015 [cit. 2016-02-19]. Dostupné z: http://redux.js.org/index.html

[5] Django. Django: The web framework for perfectionists with deadlines. [online]. 2005-2016 [cit. 2015-12-25]. Dostupné z: https://www.djangoproject.com/

[6] W3Schools – SVG Tutorial. W3Schools. [online]. © 1999-2016 [cit. 2015-12-12]. Dostupné z: http://www.w3schools.com/svg/

[7] KOPPERS, Tobias. Webpack module bundler. Webpack. [online]. 23.1.2016 [cit. 2016-02-23]. Dostupné z: https://webpack.github.io/

[8] HALLIDAY, James. Github – Browserify Handbook. Github. [online]. [2015] [cit. 2015-12-03]. Dostupné z: https://github.com/substack/browserify-handbook

[9] Github – Gulp Documentation. Github. [online]. [2016] [cit. 2016-01-23]. Dostupné z: https://github.com/gulpjs/gulp/tree/master/docs

A Nástroje pro sestavení

V případě použití Reactu nainstalovaného manažerem balíčků `npm`[8] a importováním knihovny do JavaScriptového souboru klíčovým slovem `import`, dostupným ve verzi ECMAScript 2015, nebo použitím Node.js syntaxe pro vkládání modulů, je nutné projekt nejdříve sestavit do podoby srozumitelné prohlížeči. K tomuto sestavení lze použít některý z nepřeberného množství nástrojů. V následujících kapitolách stručně popíšu tři nejpoužívanější.

A.1 Gulp

Gulp je rychlý a intuitivní nástroj využívající streamů softwaru Node.js[9]. Konfigurace tohoto nástroje spočívá ve vytváření úkolů, které berou jako argument cesty k zdrojovým souborům, provedou s načtenými daty potřebné změny a zapíší jejich výsledek do nového souboru. Změny, které s daty provádíme určují pluginy, těch je na internetu k dispozici nepřeberné množství. [9]

Pluginy mohou plnit různé funkce. Některé kompilují kódy jiného jazyka, například CSS preprocesorů na soubory s příponou .css, a jiné zase spojují sadu výsledných souborů do jednoho za účelem urychlení načítání webových stránek. Celý proces spočívá v předávání si Vinyl objektů, což je objektová reprezentace metadat zpracovávaných souborů, mezi jednotlivými pluginy, které provádí dané změny. Celý tento proces se děje v operační paměti počítače a je proto velice rychlý.

Konfigurace jednotlivých úkolů se píše do souboru `gulpfile.js` v JavaScriptu, je tedy možné používat i možnosti klasického JavaScriptu spolu s Gulp API. Jednotlivé úkoly mohou mít závislosti na jiných úkolech, je tedy možné definovat úkol, který nedělá nic jiného než, že spouští jiné úlohy pro zpracování všech zdrojových souborů naší aplikace. Gulp se snaží tyto úlohy zpracovávat paralelně k zajištění co nejrychlejšího zpracování.

```
1 gulp.task('copy-images', function () {
2   return gulp.src('src/images/**')
3     .pipe(minifyImagesPlugin())
4     .pipe(gulp.dest('dest/images/'));
5 });
```

Výpis A.1: Ukázka konfigurace úkolu pro sestavení v Gulp

A.2 Browserify

Browserify je open-source JavaScriptových nástroj, který umožňuje psát kód s využitím Node.js syntaxe pro vkládání modulů. Browserify při spuštění bere jako argument cestu k jedinému souboru, který je kořenem výsledné aplikace, a poté postupuje podle vkládaných závislostí, z kterých

[8] NPM - nástroj pro správu JavaScriptových knihoven, umožňuje instalaci konkrétních verzí, vyhledávání a aktualizaci stávajících knihoven.

[9] Node.js - softwarový systém navržen pro psaní moderních webových aplikací, především serverů, pomocí JavaScriptu, postaven na rychlém virtuálním JavaScriptovém enginu V8 od společnosti Google.

vytvoří výsledný soubor zahrnující jak zdrojové kódy, tak i požadované externí knihovny a tento soubor již může být bez problému vložen do HTML a interpretován prohlížečem. [8]

Podobně jako je tomu u nástroje Gulp i Browserify podporuje pluginy. V mé bakalářské práci například používám, mimo celkové sestavení spustitelného souboru, Babel plugin pro transformaci ECMAScript 2015 syntaxe na ECMAScript 5. Nicméně browserify streaming API funguje jinak než Gulp API, není proto možné přímo používat Gulp pluginy pro transformaci výstupního souboru. Tento problém řeší npm plugin `vinyl-source-stream`, tento plugin transformuje textový stream, které je výstupem browserify, na stream vinyl objektů, na které lze bez problému použít jakýkoliv gulp plugin.

Browserify může být spuštěn z příkazové řádky příkazem `browserify` nebo uvnitř jiných nástrojů pro sestavení po importování modulu browserify.

```
1 gulp.task('browserify', function() {
2   var bundleStream = browserify('./index.js').bundle();
3
4   return bundleStream.pipe(vinylSourceStream('index.js'))
5     .pipe(gulpUglifyPlugin())
6     .pipe(gulp.dest('./'))
7 });
```

Výpis A.2: Ukázka použití Browserify spoulu s nástrojem Gulp

A.3 Webpack

Jedná se o nejmladšího hráče na poli nástrojů pro sestavení. Zatímco Browserify se zabývá zpracováním modulů a Gulp o správu ostatních statických souborů jako například stylů nebo obrázků, Webpack se snaží řešit oba problémy. [7]

Webpack nabízí kompletní řešení pro správu statických souborů rozsáhlých moderních webových stránek, navíc zahrnuje funkce jako code splitting, to umožňuje rozdělení kódu na „kousky“, které jsou načítány na požádání, což zvyšuje celkový výkon zejména single page webových aplikací.

Webpack umožňuje v JavaScriptu vkládání nejen JavaScriptových modulů, ale také prakticky jakýkoliv jiných statických souborů. To je možné s využitím Loaderů, ty transformují jiné než JavaScriptové soubory do formátu vhodného k zpracování. Dá se tedy říct, že jakýkoliv statický soubor je zároveň JavaScriptový modul.

Webpack se spouští příkazem `webpack` z příkazové řádky, konfigurace je možná v souboru `webpack.config.js`, tento soubor musí exportovat nový modul, který je zároveň JavaScriptový objekt.

```
1 module.exports = {
2   entry:  './src',
3   output: {
4     path: 'builds',
```

```
5       filename: 'bundle.js',
6     },
7   };
```

Výpis A.3: Ukázka konfiguračního souboru nástroje Webpack

B Specifikace požadavků webové aplikace

Na základě specifikace zadání v kapitole 8.1 lze blíže specifikovat jednotlivé požadavky na vývoj této aplikace. Tyto požadavky budou dle metody FURPS[10] rozděleny do pěti kategorií zaměřujících se na jejich: funkčnost, užitečnost, spolehlivost, výkon a rozšiřitelnost.

Poznámka 3 Význam priorit jednotlivých požadavků je následující: **1** (nízká) - **5** (vysoká).

B.0.1 Funkčnost

FR1 - Přihlášení administrátora

- **ID:** FR1
- **Priorita:** 5
- **Popis:** Administrátor má možnost přihlášení za účelem správy dat.
- **Závislost:** —

FR2 - Správa map

- **ID:** FR2
- **Priorita:** 5
- **Popis:** Administrátor může pomocí dostupných nástrojů vytvářet a spravovat jednotlivé mapy.
- **Závislost:** FR1

FR3 - Správa událostí

- **ID:** FR3
- **Priorita:** 5
- **Popis:** Administrátor může spravovat události, vytvářet jejich obsah a přiřazovat jim dříve vytvořené mapy.
- **Závislost:** FR1

[10]FURPS - tato metoda byla vytvořena společností Hewlett-Packard, která se dívá na kvalitu software nebo informačního systému z pěti hledisek: funkčnost, užitečnost, spolehlivost, výkon a rozšiřitelnost.

FR4 - Poskytnutí veřejného API

- **ID:** FR4
- **Priorita:** 4
- **Popis:** Webová aplikace bude poskytovat veřejné API pro přístup k datům z mobilní aplikace.
- **Závislost:** —

FR5 - Generování QR kódů

- **ID:** FR5
- **Priorita:** 3
- **Popis:** Webová aplikace bude poskytovat jednoduché nástroje pro generování QR kódů s odkazem na jednotlivé akce jako jsou: pozice na mapě, specifické události nebo soutěže.
- **Závislost:** —

B.0.2 Užitečnost

UR1 - Snadná ovladatelnost a konzistence UI

- **ID:** UR1
- **Priorita:** 4
- **Popis:** Ovládací prvky celého UI budou konzistentní, přehledné a jednoduché tak, aby umožnily snadné ovládání pro všechny typy uživatelů.
- **Závislost:** —

B.0.3 Spolehlivost

RR1 - Kontrola vstupů

- **ID:** RR1
- **Priorita:** 3
- **Popis:** Aplikace by měla v případě neplatných vstupů či jiných chyb v rámci zpracování požadavků vždy zobrazit chybová hlášení.
- **Závislost:** —

B.0.4 Výkon

PR1 - Minimalizace počtu požadavků zasílaných na server

- **ID:** PR1
- **Priorita:** 2
- **Popis:** Aplikace by měla minimalizovat počet požadavků zasílaných na server z důvodu snížení zátěže serveru.
- **Závislost:** —

PR2 - Přenesení zátěže na klientův počítač

- **ID:** PR2
- **Priorita:** 3
- **Popis:** Operace jako validace formulářů a kreslení mapy budou přeneseny na klientův počítač s využitím JavaScriptu a knihovny React.
- **Závislost:** —

B.0.5 Rozšiřitelnost

SR1 - Schopnost rozšíření

- **ID:** SR1
- **Priorita:** 3
- **Popis:** Aplikace by měla být navržena s ohledem na možnou rozšiřitelnost.
- **Závislost:** —

C Specifikace požadavků mobilní aplikace

Na základě specifikovaného zadání v kapitole 10.1, můžeme blíže určit jednotlivé požadavky na vývoj mobilní aplikace. Podobně jako tomu bylo u specifikace požadavků webové aplikace, budeme požadavky třídit dle metody FURPS do pěti kategorií.

Poznámka 4 Význam priorit jednotlivých požadavků je následující: **1** (nízká) - **5** (vysoká).

C.0.1 Funkčnost

FR1 - Zobrazení událostí

- **ID:** FR1
- **Priorita:** 5
- **Popis:** Mobilní aplikace bude zobrazovat seznam událostí získaný z webového API.
- **Závislost:** —

FR2 - Zobrazení mapy

- **ID:** FR2
- **Priorita:** 5
- **Popis:** Mobilní aplikace bude v rámci okna událostí zobrazovat přiřazenou mapu.
- **Závislost:** FR1

FR3 - Podpora akčních objektů na mapě

- **ID:** FR3
- **Priorita:** 5
- **Popis:** Mapy daných událostí budou podporovat akční objekty, které po kliknutí zobrazí další informace.
- **Závislost:** FR2

FR4 - Podpora čtení QR Kódů

- **ID:** FR4
- **Priorita:** 3
- **Popis:** Mobilní aplikace bude obsahovat modul pro čtení námi vygenerovaných QR kódů poskytujících další funkčnost.
- **Závislost:** —

FR5 - Ukládání postupu hry (soutěže)

- **ID:** FR5
- **Priorita:** 3
- **Popis:** Mobilní aplikace bude podporovat ukládání postupu her (soutěží), kterých se uživatel účastní přečtením patřičných QR kódů.
- **Závislost:** —

C.0.2 Užitečnost

UR1 - Konzistence uživatelského rozhraní

- **ID:** UR1
- **Priorita:** 5
- **Popis:** Jednotlivé prvky uživatelského rozhraní mobilní aplikace budou přehledné, snadno ovladatelné a jejich vzhled bude konzistentní v rámci dané platformy.
- **Závislost:** —

C.0.3 Spolehlivost

RR1 - Stabilita

- **ID:** RR1
- **Priorita:** 4
- **Popis:** Mobilní aplikace bude korektně spravovat chybové stavy, které mohou nastat v rámci jejího chodu.
- **Závislost:** —

C.0.4 Výkon

PR1 - Minimalizace přenášených dat

- **ID:** PR1
- **Priorita:** 3
- **Popis:** Mobilní aplikace bude cachovat data získaná z webového API za účelem minimalizace dat přenášených po síti.
- **Závislost:** —

C.0.5 Rozšiřitelnost

SR1 - Schopnost rozšíření

- **ID:** SR1
- **Priorita:** 4
- **Popis:** Mobilní aplikace by měla být navržena s ohledem na možnou rozšiřitelnost v rámci dostupných API.
- **Závislost:** —

Přílohy na přiloženém DVD

Adresářová struktura příloh na přiloženém DVD je následující:

- **Příloha D** - Zdrojové kódy webové aplikace
- **Příloha E** - Zdrojové kódy mobilní aplikace
- **Příloha F** - Android aplikace ve formátu .apk